Integrating Design Decision Management with Model-based Software Development

Könemann, Patrick

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

Integrating Design Decision Management with Model-based Software Development

Patrick P. Könemann

Kongens Lyngby 2011
IMM-PHD-2011-249

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

Design decisions are continuously made during the development of software systems and are important artifacts for design documentation. Dedicated decision management systems are often used to capture such design knowledge. Most such systems are, however, separated from the design artifacts of the system. In model-based software development, where design models are used to develop a software system, outcomes of many design decisions have big impact on design models. The realization of design decisions is often manual and tedious work on design models. Moreover, keeping design models consistent with all made decisions is a time-consuming manual task that is often performed in peer reviews.

In this thesis, a generic technology has been developed for extracting model differences from models and transferring them to other models. These concepts, called model-independent differences, can be used to specify realizations of decisions in design models. This way, recurring realization work of design decisions can be automated. Since the concepts are generic and not bound to design decisions, other recurring work on models can be automated as well, for instance, design patterns and refactorings.

With such a technology at hand, design decision realizations can easily be specified and parts of the realization work can be automated. A binding is produced as a by-product that links documented decision outcomes to design model elements which are affected by the respective decisions. With a set of constraints, such a binding can be used to validate the consistency between the design and made design decisions. Whenever the evolving design models become inconsistent with realized decisions, developers are notified about the violations. The

violations can be fixed by correcting the design, adjusting the binding, or by ignoring the causes. This substitutes manual reviews to some extent.

The concepts, implemented in a tool, have been validated with design patterns, refactorings, and domain level tests that comprise a replay of a real project. This proves the applicability of the solution to realistic examples. The implementation of model-independent differences, called MPatch, is further contributed to the Eclipse open source project.

Resumé

Ved udviklingen af software systemer er der taget design beslutninger, som er vigtige artefakter til design dokumentation. Dedikeret beslutningsmanagementsystemer bruges ofte til at dokumentere design viden. De fleste værktøjer er dog adskilt fra designet af systemet. I model-baseret software udvikling har resultaterne af mange design beslutninger stor indflydelse på design modellerne. Normalt kræver det manuelt og besværligt arbejde at realisere design beslutninger i design modellerne. Desuden er det en tidskrævende manuel opgave, at holde design modeller i overensstemmelse med alle trufne beslutninger, som ofte foretages i peer reviews.

I denne afhandling blev der udviklet en teknologi til specifikation af realiseringer af design beslutninger. De koncepter, der kaldes model-independent differences, omfatter en generisk teknologi til udvinding af forskelle fra modeller og overføre dem til andre modeller. På denne måde kan gentagne realiseringer af design beslutninger automatiseres. Da koncepterne er generiske og ikke bundet til design beslutninger, kan også andet gentagende arbejde på modeller automatiseres, for eksempel design patterns og refactorings.

Med sådan en teknologi ved hånden kan realiseringer af design beslutninger nemt specificeres og noget af realiserings arbejdet kan automatiseres. En binding fremstilles som et biprodukt, som forbinder dokumenterede beslutninger med model elementer, der er berørt af de respektive beslutninger. Med et sæt af begrænsninger kan sådan en binding bruges til at validere sammenhængen mellem design og realiserede beslutninger. Når design under udvikling bliver uforeneligt med trufne design beslutninger, bliver udviklerne underrettet herom. Uoverensstemmelser kan rettes ved at korrigere design, tilpasse binding eller ignorere årsagerne. Denne metode erstatter manuelle reviewes i vid udstrækning.

Disse koncepter, som er blevet implementeret i et værktøj, blev valideret med design patterns, refactorings og domain level tests, som omfatter en replay af et reelt projekt. Dette beviser anvendeligheden af løsningen til realistiske eksempler. Implementeringen af model-independent differences, kaldet MPatch, er yderligere bidraget til open-source projektet Eclipse.

Preface

> "Software documentation is a waste of time!"

This was one of the statements I was making in the early days as a programmer. All programs I was confronted with these days were at most 1000 lines of source code, which can indeed be understood without documentation. Today, after having participated in software development teams between a half and 30 man-years, I have a different opinion on software documentation. Without properly documenting a piece of software, it is very hard, for instance, for new team members and maintainers to understand a software system, especially when it continuously grows. Understanding such software and researching the rationale behind a design then requires a lot of effort. This is why I revised my opinion to the following provocative statement.

> "If a software is not documented, it does not exist!"

This, of course, only applies to reasonably complex software in which the time for understanding it might exceed the time for rewriting the software. Since the pure documentation task of software is tedious and without immediate benefit for most developers, one of my goals for this project is to provide developers a method in which they immediately benefit from documenting their software. There are several tradeoffs I was confronted with in order to reach that goal. I would like to briefly reflect on two of them.

A tradeoff that I often discuss with others concerns the degree of automation. *"This is always the automation I was looking for!"* vs. *"such a sensible step should never be automated – developers must know what is going on here!"*

I believe that there is truth in both perspectives. For the concepts and the prototype I developed, it was about finding the right balance between providing enough automation that developers do not need to do recurring tasks and, at the same time, providing enough control and transparency to developers to understand and steer the tasks.

This brings me to the next tradeoff concerning functionality vs. usability. *"It might even be the coolest feature ever, but if it is too complex to use, no one will use it!"* Unfortunately, the circumstances under which I developed the prototypic tool in the past three years, did not involve others who could help with this task. Therefore, I had to concentrate on functionality with the result of decent usability. If the developed tool shall be used beyond an academic evaluation of the proposed concepts, the user interface needs some improvements to make the new features easily accessible to the average software developer. Still, I hope that I can convince many developers that documenting software is not anymore a tedious task without immediate benefits.

The five research papers below have been published at conferences and workshops in the last three years whose contents are incorporated into this thesis.

[Kön10a] Patrick Könemann. *Capturing the Intention of Model Changes.* In Proceedings of the 13th International Conference of Model Driven Engineering Languages and Systems (MoDELS), October 2010.

[KZ10] Patrick Könemann, Olaf Zimmermann. *Linking Design Decisions to Design Models in Model-based Software Development.* In Proceedings of the 4th European Conference on Software Architecture (ECSA), August 2010.

[Kön10b] Patrick Könemann. *Semantic Grouping of Model Changes.* In Proceedings of the 2010 TOOLS International Workshop on Model Comparison in Practice (IWMCP), July 2010.

[Kön09b] Patrick Könemann. *Integrating Decision Management with UML Modeling Concepts and Tools.* In Working Session of the Joint Working IEEE/ IFIP Conference on Software Architecture 2009 (WICSA) & European Conference on Software Architecture (ECSA), September 2009.

[Kön09c] Patrick Könemann. *Model-Independent Differences.* In Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM), May 2009.

This thesis was prepared at the department of Informatics and Mathematical Modeling, Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering.

Lyngby, May 2011

Patrick P. Könemann

Acknowledgements

I thank my supervisor Assoc. Prof. Dr. Ekkart Kindler for an exceptional supervision and flawless guidance of my Ph.D. project and for the countless and enlightening discussions despite busy schedules. Thank you for all the support during the fabulous three years.

I thank my colleagues at the Software Engineering section at DTU and at the University of Paderborn for numerous helpful discussion. I also thank all the people I met and had discussions with during conferences and workshops.

I thank Dr. Olaf Zimmermann for all the creative discussions we had, especially for his suggestions and encouragements, and for his kind hospitality during my visits at the IBM Research Lab in Zürich.

In addition, for reading parts of an early draft of this thesis, I thank Markus von Detten, Christian Gerth, Stefanie Knust, Anna-Lena Meyer, Jan Rieke, Dietrich Travkin, and Olaf Zimmermann.

I thank the EMF Compare team for their interest in my work, for helping with technological problems, and for integrating MPatch into the project's repository.

It has been a pleasure to be part of DTU Informatics the last three years. In particular, I enjoyed plenty of coffee breaks with Julia Borghoff and Michael Reibel Boesen which distracted me from work and made me gain new energy.

Last but not least, I thank Stefanie Knust and my family for supporting me, in particular, when things were not going as expected.

Glossary

Alternative. An alternative is a possible solution of an issue for a decision; see also Decision.

API. An Application Programmable Interface of an application allows communication and interaction with other tools.

(Software) Architect. The person who is responsible for the architecture of a software system.

Architectural (Design) Decision. An architectural design decision is a special kind of a design decision that concerns the architecture of the software system.

(Software) Architecture. This is the global structure of a software system, often described as a set of components and connectors between them; nonrelevant technical details or inner structures of components are usually omitted.

(Decision) Binding. A decision binding or binding is a link between the outcome of design decisions and particular elements in the design artifacts of a software system.

Decision. A decision is the choice of a solution for a particular problem; possible solutions are called alternatives and the instance of a decision is called outcome; see also Design Decision.

Decision Maker. The person who is responsible for making decisions.

(Software) Design. The design of a software system includes the higher-level architectural design and its refinements; this may also include technical and platform-specific, details.

Design Decision. This is a decision that has an effect on the design of a software system during its development; see also Decision.

Design Documentation. It contains details about the logical and technical design of a software system, including structural and behavioral parts and ideally also the rationale behind the design.

Design Issue/Problem. A design issue defines a particular issue or problem in the design of a software system.

Design Model. The design is described in terms of a model such as a UML model.

Design Model Changes/Updates. Design model changes or design model updates are made by software developers whenever they work on or extend the design model; model differences can be used to calculate and describe such changes.

Design Pattern. A design pattern is a widely accepted solution to a recurring design problem; the description often includes the problem, the solution, variants, examples, and relations to other patterns.

(Software) Developer. A person who participates in the development of a software system.

Metamodel. A metamodel is the model of a model which defines the structure of a model, an example is the UML metamodel; see also MOF.

Model. A model is the abstraction of a software system, neglecting mostly technical details; it may consist of multiple parts, each focusing on one particular aspect of the system.

Model Change. A model change describes a modification in a model; examples for UML models are added elements like classes and activities, updated properties like the names of elements, and removed references like inheritance relations.

Model Differences. This is a set of model changes, typically calculated from two versions of a model; see also Model-dependent and Model-independent Differences.

Model Element. This is one particular element in a model; in case of UML it might be a class or an attribute in a class diagram, or an event in a sequence diagram.

Model-based Software Development. A branch of software development in which formal models are used for describing the software system, for example, to generate code out of it.

Model-dependent Differences. This is a concrete set of model changes for a pair of models that has been compared with each other; the changes refer to the compared models and cannot be applied to other models.

Model-independent Differences. This is a set of model changes that is independent of any concrete model, including the models from which the differences have been calculated from; model-independent differences are in particular also applicable to other models.

(Software) Modeler. A person who is responsible for or participates in the development of the design model of a software system.

MOF. The Meta Object Facility defines meta levels for the definition of models; M0 specifies the data in a running system, M1 a model, for instance, a design model, M2 the metamodeling language, and M3 the meta metamodeling language.

MPatch (upper case). This is the tool implementing model-independent differences.

mpatch (lower case). This is an artifact containing model-independent differences.

(Decision) Outcome. An outcome is the result of a design decision, including in particular the rationale of the decision and information about the affected parts in the design; see also Decision and Binding.

Refactoring. Structural changes of an artifact like code or a model without altering the functionality of the artifact; refactorings are typically used to clean up the artifact and/or to improve readability and extendibility.

UML. The Unified Modeling Language is a general-purpose modeling language for software systems; it contains several types of diagrams to describe structural and behavioral parts of a system.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The development of software systems is a big challenge and will become an even bigger challenge in the future because the complexity of software systems increases continuously. Especially big companies like banks, automobile manufacturers, or telecommunication providers need sophisticated and customized software, adjusted to their specific needs. Such software systems are not developed by individuals but in teams, so that different tasks are distributed among the team members. The development of a complex software system might take a long time, up to several years. The success of such a development project depends on proper coordination of the development team, tools for planning, implementing, and analysing software systems, communication between different stakeholders, and proper documentation, to name just a few factors.

An important type of documentation in ongoing research of software architecture during the last decade is design decisions. Any software developer and in particular software architects make decisions that influence the architectural design of software systems. The design of a software system comprises all artifacts that describe the design, code, models, and other artifacts describing the system. The choice of technologies, databases, or architectural and other design patterns are examples for design decisions. The outcome of a design decision has an impact on the design of the software system, hence the name *design decisions*. Jansen and Bosch present an extreme point of view in which the architectural design *consists* of design decisions [JB05, p. 1]:

> "We define software architecture as the composition of a set of architectural design decisions."

The advantage of documenting design decisions instead of only documenting the resulting design is that also the rationale behind the design is captured. This makes the design documentation better understandable to others, for instance, during software maintenance. Decision management systems help developers to document such decisions. However, the state of the practice is that design decisions are rarely documented explicitly. Their documentation is often not mandatory and takes too much time, nor is their documentation properly supported by any mature software development tool. The reason for not documenting them is usually that developers do not see immediate benefit for the extra effort. Instead, only the final design without the rationale behind it makes its way into the documentation. That makes it hard for others to follow and understand the design.

One particular field of software development uses models for the description of software systems. In this field, known as model-based software development, models with defined syntax and semantics are used to abstract from technology-specific implementations. When such models are used for developing the logical design and architecture of a software system without considering technical details, they are called design models. They can also be used for documenting the system, especially when they have a graphical representation. Since design decisions have an impact on the design, design models are also affected, and, thus, the same problem occurs: without proper documentation, design models might be hard to understand for people who did not create the design. If the rationale behind the decisions or, even worse, the decisions themselves, are not documented, the design knowledge vaporizes eventually. Another big problem is to check whether made design decisions are properly realized in design models and that the design is consistent with the documentation. Especially when decisions are not documented properly, a conformance check between design decisions and design models is a tough and tedious job to do, if possible at all.

This thesis analyses design decisions from the perspective of model-based software development and proposes methods for supporting developers with respect to the aforementioned problems. So far, design decisions have been studied on the architecture level and on linking them to code, but only little research has been performed relating design decisions to models in model-based development of software systems. Two major aspects are discussed in this work: the first is an analysis of how software developers can be supported in making and realizing design decisions. This involves the consideration of alternative solutions for open decisions, the actual documentation of made decisions, and in particular the realization of the chosen solution in the design models. If the solution is

realized in the design, we speak about realized decisions. The second aspect is how design decisions are used after they have been made and realized. That includes using them for documentation, for visualizing affected parts of the design models as well as for analyzing whether the design models and the outcomes of design decisions conform to each other.

The overall goal is to improve support for design decisions in model-based software development, structured in three subgoals. Frist, the documentation of design decisions shall be related to design models that are affected by the respective decisions. This way, design decisions shall be linked to the model. Second, the realization of recurring design decisions shall be easier and less error-prone. If design decisions recur in the same or similar projects, their solutions and realizations in the design might be automated. Third, the consistency between made design decisions and affected design models shall be validated automatically.

In this project, tool-independent concepts have been developed for achieving these goals. As a proof of concept, a prototypic tool realizes these concepts to show the technical feasibility of the goals. It integrates with a modeling tool and a decision management system as a central place for storing and maintaining design decisions and for creating a knowledge repository.

The remainder of this chapter briefly introduces the main areas the thesis is dealing with: model-based software development, decision making, and model changes as an approach for describing design templates for models. Lastly, it outlines the research methodology and contributions.

1.1 Model-based Software Development

This section introduces model-based software development as it is required to understand the contributions of the thesis, because design models are the artifacts in which the outcomes of design decisions are realized. After describing the characteristics and use of models, the next sections discuss what design decisions are, how they are made, and how extensions and changes in models can be expressed.

Models are used for several reasons to describe software systems. For many developers the most important reason for using models is raising the level of abstraction. Complex software systems can be described by models ignoring technical or other details that are not relevant when the models are used. Moreover, models offer a common and technology-independent language for developers to

discuss software designs. For some models, the syntax and semantics are well-defined and parts of the code can automatically be generated from such models.

There are different types of models for different purposes. Structural models are typically used to describe the static structure of a software system or of parts of it. Behavioral models, in contrast, can be used for describing the behavior and, thus, the dynamics of a software system. The Unified Modeling Language (UML), for instance, is a general-purpose modeling language for software systems that is widely used in academia and in industry. It includes notations for structural as well as for behavioral aspects of a software system. More specialized modeling languages are the Business Process Model and Notation (BPMN) for describing business processes, the Systems Modeling Language (SysML) as an extended subset of UML for systems engineering, or Petri Nets that can be used for simulations and software verification. The model-driven architecture approach (MDA) specifies further how such formal models can be used to stepwise refine a system specification down to the code level. This thesis focuses on structural models because it would be too difficult to also consider the semantics of behavioral models. The UML is used to illustrate examples.

Sometimes special non-functional requirements apply to a software system such as safety and security requirements, real-time behavior, or other domain-specific aspects. Such requirements entail special constraints on the design that are omnipresent during the entire development. These cases are out of scope in this thesis and it is the responsibility of the developers to properly integrate such additional requirements and their implied constraints into the design.

1.2 Decision Making in Software Development

Decisions are made in any genre of software development, not only when models are used. This section briefly explains what design decisions are, by whom they are made, and what consequences arise when making them implicitly or explicitly. Then the focus shifts towards model-based software development as a special genre of software development.

Decisions can occur in every phase of a software development project. At the beginning, the team size and members must be chosen, the development technologies must be decided, and the team must agree on one or many modeling and/or programming languages. Further typical decisions are the architectural style of the overall software system and individual components, which databases and communication protocols should be used, whether third-party application should be bought, etc.

Many decisions refer to the design of the system, like choices concerning the system architecture, databases, communication protocols, or the use of design patterns. Only decisions concerning the design of a system are relevant, organizational, strategic, and other decisions are disregarded here. Whenever we talk about design, we refer to it as a set of design artifacts. Kruchten and Kroll [KK03] define an artifact as one of many kinds of tangible by-products produced during the development of software. A design artifact can, for instance, be design documents, design models, test data, or code. We call any decision that has direct or indirect impact on design artifacts a *design decision*; the result of such decisions will be reflected in design artifacts. The meaning of indirect impact is that the decision is related to the design and subsequent work or decisions will modify the design.

The overall task of making design decisions can be divided into three steps: first, the identification of a design issue, e.g. the need for an architectural style; second, the choice of a particular solution, e.g. to use a layered architecture; third, the realization of that solution in the design, e.g. to create appropriate components in a design model. Depending on the severity and impact of the decision, the selection of the solution and its realization might be performed by people with different roles. Software architects are the main decision makers for significant decisions and their realization in the design is mostly delegated to developers like modelers or programmers, whoever is suited best. This may also be the same person acting as both roles. Before explaining that delegation, we briefly discuss the consequences of decisions whose outcomes are not documented.

In small or short-term projects, the documentation of most design decisions might not be necessary because asking other developers is often fast and easy. Undocumented and unconsciously made decisions are implicit decisions – in contrast to explicit decisions that are documented and, thus, consciously made. In big and long-term projects, team members come and go, developers might forget what they have developed a year ago, and, consequently, the undocumented rationale behind the design vanishes eventually. New team members and maintenance engineers have a tough job understanding the system properly because of outdated or missing documentation. Hence, they either have to spend much time understanding the design or they might add errors because they are unable to estimate the impact of their modifications. In order to avoid these scenarios, the project management or lead architects could enforce an explicit documentation policy for the design and decisions affecting it.

After a design decision has been made, the chosen solution might require an extension or change of the current design of the system. Assuming that a software architect made the decision, he or she might delegate the realization task to other developers. This delegation is usually carried out face-to-face or with

an example solution and a description that show how the solution can be realized in related design artifacts. A major part of this thesis discusses how such realizations of design decisions and their implied changes in and extensions of design models can be specified and automated. To this end, the next section introduces model changes and how they can be described.

1.3 Describing Model Changes

Most design decisions imply changes in design artifacts which, in case of model-based software development, are models. Considering design decisions that are frequently made, like the realization of design patterns, the implied changes in the design are recurring work every time this particular decision is made. This section introduces the concepts of model changes for specifying such decision realizations, similar to a template that can be applied to arbitrary design models. That way, the realization of design decisions can, to some extent, be automated.

The naive way of describing changes in any model is to express what has been removed and what has been added. However, we might lose important information like the identity of particular model elements, for instance: moving some element in a model to another place has a different meaning to the user than deleting that element in the old place and adding some element at the other place. It would be difficult for the user to see that an addition and a deletion express a move, in particular if these are only two within a big set of model changes. So we need adequate concepts to describe meaningful model changes that the user understands.

All aforementioned types of models, including UML, can be represented as attributed graphs which distinguish between nodes, references between nodes, and node attributes. There are at least nine types of changes for attributed graphs to cover all kinds of changes for such models: the addition, deletion, and update of model elements (nodes), references (edges), and attributes. Fig. 1.1 illustrates three of these changes on exemplary UML model fragments. The left-hand side of Fig. 1.1 illustrates the deletion of an element, the UML interface *IController*. The middle part depicts an added reference, an inheritance relation between two classes. The right-hand side indicates an update of the attribute *isAbstract* which is defined for all classifiers in UML.

The outcomes of design decisions extend or change existing parts of a design model. Such extensions or modifications could be described with a set of model changes like the ones just presented. When a user wants to extend an existing model by realizing a design pattern, which is the outcome of a design decision,

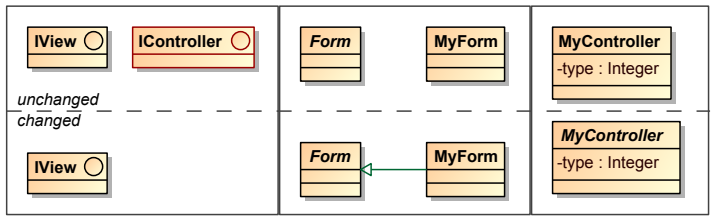


Figure 1.1: Example changes; left: a deleted element (node in UML: *interface*); middle: an added reference (reference in UML: *generalization*); right: an updated attribute (attribute in UML: *isAbstract*).

that realization has probably been performed before by someone else in another project. Suppose that this realization is available as a set of prescribed model changes, and that these model changes can be applied to the user's model, we can save time because there is no need anymore to realize the pattern by hand. Moreover, the automated application of predefined model changes reduce the likeliness to introduce modeling mistakes.

There are two large areas in the literature that deal with describing model changes: model differencing technologies compare existing models and calculate and visualize differences; model transformation technologies modify existing or create new models based on a transformation specification. Their concepts intersect to some extent, but their intentions are different. Model transformations are mostly non-interactive and come with their own specification language and notation for transformation rules. Model differencing, in contrast, is usually interactive and uses the notation of the model of interest, but the computed model changes are typically not applicable to other models than the compared ones. Although both kinds of technologies have their advantages, neither of them is properly capable of describing, applying, and persisting design model changes. Section 5.1 compares the two technologies in more detail.

1.4 Research Methodology

This research project tackles existing problems in literature about design decision support in software development. In contrast to existing work, we concentrate on model-based software development and focus on design decisions that relate to design models. Here we give a high-level overview of the motivation, the goals, and the proposed solution.

Motivation. Existing work deals with the documentation, capturing, making, and reasoning of design knowledge including design decisions. However, the support of design decisions in model-based software development is unsatisfactory, most concepts relate decisions to source code. Modeling tools and decision management systems merely coexist without proper integration. Thus, the ultimate motivation is to achieve an integration of design decisions with design models similarly as it is already the case for source code.

Goals. The goal in this research project is an improvement of design decision support in model-based software development in respect of the following aspects.

1. The documentation of design decisions shall be related to design models which are affected by the respective decisions. A link between these two artifacts, design decisions and design models, shall improve the documentation of design decisions and design models.
2. The realization of design decisions shall be easier and less error-prone. Design templates for recurring design decisions shall help developers to automatically realize design decisions. The creation and application of such design templates shall be easy and with little effort.
3. Realizations of design decisions in design models shall be validated automatically.

Solution. We propose a model differencing technology to specify design decision realizations as design templates and, in this way, integrate design decisions with design models. This addresses the aforementioned subgoals which have not yet been solved for model-based software development. Our solution extends and adjusts existing use cases for handling design decisions to the new setting. A prototype was developed that implements the conceptual solution and shows the technical feasibility of the extended and adjusted use cases.

Evaluation. An evaluation of the proposed solution would at least consist of a feasibility study and an experiment in which the concepts are applied in a realistic environment like a running project. The first part of our evaluation is a prototypic implementation of the proposed concepts including correctness and interoperability tests. Due to time constraints of this research project, the prototype was not applied in a running project. Instead, we performed several applicability and domain level tests to simulate realistic examples. A case study in which the prototype is applied in a running project remains as future work.

1.5 Scope

The solution presented in this thesis includes new and adjusted use cases for design decision support in model-based software development as well as a new model differencing technology. This section discusses the scope, important assumptions, and limitations of the presented solution.

There exists already a lot of research about design decision support and the literature lists plenty of use cases which support developers in, amongst others, identifying, making, documenting, and analyzing design decisions. Instead of reinventing the wheel, this thesis assumes that existing use cases and tools can be extended and complemented to fit the context of model-based software development. We focus on design decisions that are closely related to design models. Many other approaches provide support for decisions concerning, for instance, the development process of a project or its organization. Our solution allows to document such decisions as well, but reuse of design decision solutions is limited to decisions which imply concrete changes in design models. Hence, the main purpose of the model differencing technology is to specify and automate changes in design models that are implied by fine-grained design decisions. Since changed model elements are later traced to design decisions and vice versa, our solution works best if the number of changed model elements per decision is concise and in total at most 20–30, depending on how the complexity of all changes can be understood by developers.

Examples for model changes of interest are refactorings, application of design patterns, or any other concrete modifications on design models. The differencing technology defines model changes in terms of the models' metamodels, and some of the presented concepts require textual attributes on model elements. Hence, models must be defined in terms of a metamodel and individual for the generalization of model changes, model elements must contain textual attributes.

We further assume that model differences created from one particular type of models shall again only be applicable to the same type of models. For instance, if model changes were specified for UML models, it is sufficient that they are again only applicable to UML models. This is reasonable in most cases, since recurring design decisions are typically applied in the same context.

1.6 Contributions

The previous sections introduced the two major areas of this research project, decision making in software development and model-based software development. The contributions are divided into conceptual work, two concrete frameworks, and the lessons learned from the validation of these frameworks.

The first contribution refers to the first goal and presents an analysis and concepts of how decision making tasks and their documentation can be integrated into model-based software development. A tool implements the concepts. This includes an interface description for the interoperability between decision mangement systems and modeling tools. Benefits are easy-to-access documentation of design decisions for design models and reuse of design knowledge.

The second contribution are concepts and tool support for specifying, creating, and applying realizations of design decisions in design models. These technical concepts are called model-independent differences and refer to the second goal. Benefits are the automation of redundant, recurring, and tedious work on design models as well as the consideration of alternative solutions. The latter requires that realization specifications of design decisions can be stored, for instance, in a knowledge repository.

The third contribution are concepts and tool support for validating that design models conform to made design decisions. This refers to the third goal and is achieved by linking the outcomes of design decisions to the affected design model elements. Based on these links, a set of constraints ensures that the models conform to all realized decisions. The benefit is an early detection of potential modeling errors and inconsistencies.

Two tools have been developed and published as part of this thesis. First, *MPatch* is an implementation of model-independent differences and has been integrated into EMF Compare, part of the open source project Eclipse. Example applications are capturing refactorings or design patterns and applying them to other models. Second, the framework for *Design Decision Support* in model-based software development integrates decision management systems with model-based development tools. It realizes the concepts of the first and third contributions, makes use of MPatch, and uses external decision management systems to connect to a design knowledge repository.

Chapter 2

Example

The example in this chapter introduces the domain of making decisions in software development. It is an excerpt from a bigger example [Kön09a] and illustrates two decisions for the logical design in the model-based development of a fictitious enterprise application. In this chapter, the example is explained without any new concepts that are introduced in this thesis. The remaining chapters refer to this example and Chapter 6 extends it to illustrate concepts.

Both design decisions in this example originate from patterns from the book *Patterns of Enterprise Application Architecture* [Fow02]. The decisions are made by an architect whereas the realizations of their outcomes are performed by other developers. Before diving into the decision making tasks, a fictitious background and preceding decisions are outlined.

2.1 Background

A fictitious enterprise E sells and installs devices for end-users for measuring their energy consumption. E's agents visit the end-users for installing the devices, for taking a reading from them, and for maintaining them. To this end, the agents connect to a central server to access and update customer data and the readings.

Let us assume we develop the server software for enterprise E which shall provide the customer data to the agents. We will develop this enterprise software in a team consisting of a project lead and several developers, including one software architect and several modelers and programmers. Let us further assume that the design decisions below have already been made. It is irrelevant at this point when and by whom these decisions were made, they are only listed here to introduce the setting to the reader.

- A client-server architecture will be used, having one central enterprise server and one client per agent.
- The communication between the agents and the server will be web-based via the internet.
- Three layers will be used on the server to separate the data (*data access* layer), the business logic (*domain* layer), and the communication between the server and the agents (*presentation* layer).
- The presentation layer is organized according to the MVC (Model-View-Controller) paradigm.

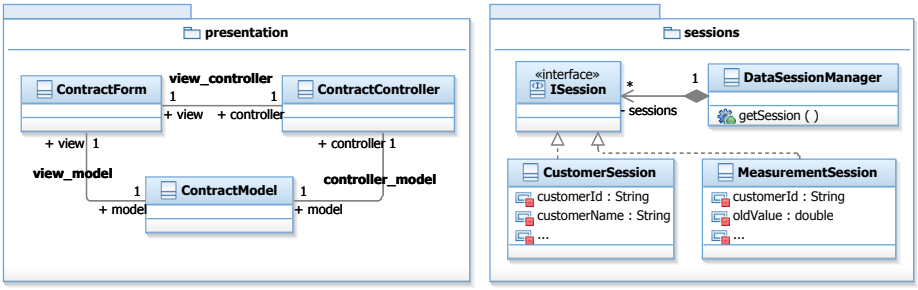


Figure 2.1: An excerpt of the presentation layer in the design model: the package *presentation* comprises components according to the MVC paradigm, *sessions* contains several components for session handling.

Let us assume that a design model already exists which incorporates all of these design decisions. The two design decisions presented in the next section refer to the presentation layer depicted in Fig. 2.1. Three classes on the left-hand side represent the respective components of the MVC paradigm; package *sessions* contains several classes for handling sessions in other parts of the application. This model is a conceptual design model and not yet a low-level implementation model.

2.2 Design Decisions

We will now make two design decisions with the aid of a decision template. A decision template enforces decision makers to prepare a decision before it is made and forces them to properly document it afterwards. For better readability, the template in Tab. 2.1 is a simplified version of what is used in literature [SLK09].

Preparation:	**Documentation:**
• What is the problem to be solved? • What are the decision drivers? • For each potential solution: – Description of the solution – Advantages and disadvantages	• Which is the chosen solution? • Justification of that choice • Consequences of this decision

Table 2.1: A simplified template for making a design decision; the questions in the left column should be prepared before the decision is made, the questions in the right column should additionally be answered for documentation.

The first of the two following decisions that is made by the software architect specifies whether or not to store the session during the communication between the server and the agents. Sessions are useful if the agents work on web forms that consist of several pages; then the data is stored in a session until submitting it to the business logic of the server. Advantages are better usability and optimized communication between server and client. The downside is a more complex design for proper session handling. Let us assume the architect decides to use sessions, then the decision could be documented as shown in Tab. 2.2. The design issue is listed in the first column, possible solutions in the second, and the result of the decision in the third column.

Design issue	**Possible alternatives**	**Decision outcome**
Session Awareness: whether or not to store sessions during the communication with clients *Decision drivers:* client-server communication, performance, usability	**Yes:** the session is stored; this allows complex communication between a client and the server. Only reasonable if there are complex transactions, e.g. multi-page forms or similar scenarios in which data is collected. *Follow-up decision:* Session Management. **No:** the session is not stored; only reasonable for simple communication or where session data may be volatile.	*Choice:* **Yes.** *Justification:* maintenance and updates of customer data is complex and involves multiple pages on a web form. *Consequences:* consider issue *Session Management.*

Table 2.2: Documentation of the design decision *Session Awareness* with the alternatives *Yes* and *No* and the outcome *Yes*.

Although this decision has important consequences on the design model, namely whether or not it should use sessions for the communication between server and clients, it does not yet imply any immediate changes in it. Instead, the outcome of the follow-up decision *Session Management* affects the design model. Fowler presents three patterns that realize session management with different advantages and disadvantages. The middle column in Tab. 2.3 summarizes these patterns.

Design issue	Possible alternatives	Decision outcome
Session Management: realization of a session and its maintenance.	**Client Session State:** the client is responsible for creating and maintaining sessions. **Server Session State:** the server is responsible for creating and maintaining sessions. **Database Session State:** session data will be stored in a database and queried on each request.	*Choice:* **Server Session State.** *Justification:* only few clients are expected and sessions can easily be handled on the server; a session manager already exists. *Consequences:* proper maintenance of a session is required; the existing session manager should be adequate and may be extended, if required.

Table 2.3: Documentation of the design decision *Session Management*.

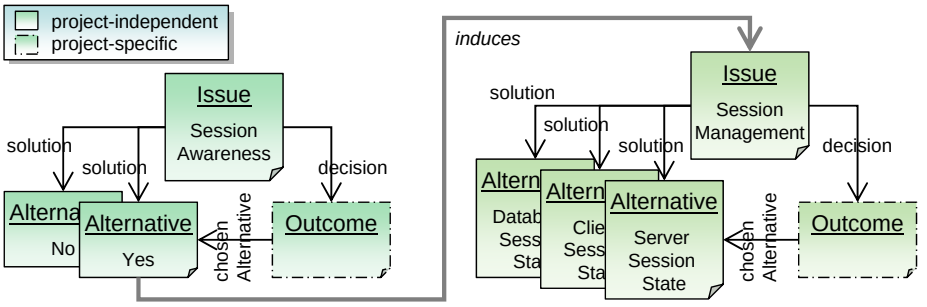


Figure 2.2: A simplified overview of both decisions *Session Awareness* and *Session Management* from Tab. 2.2 and 2.3 and a relation *induces* between them.

The architect decides to use the server session state pattern for two reasons: first, only few clients are expected so the server can easily handle session management; second, a session manager already exists on the server and, thus, the realization effort is expected to be small. Figure 2.2 gives an additional overview of both design decisions and their relation. The issues, alternatives, and decision outcomes are the same as in Tab. 2.2 and 2.3. The relation *induces* links to the follow-up decision *Session Management*.

In the next step, the architect delegates the realization task to another developer. Besides an explanation of the decision and its outcome, the architect provides a reference to the pattern definition, an example realization, and a brief description of how the pattern should be integrated with existing components. The resulting part of the design model is shown in Fig. 2.3 after the decision has been realized: a new class *ContractSession* and appropriate associations to and between existing classes were added by the other developer. In addition, he or she adds a textual description to the decision outcome and the design documentation about which model elements have been changed due to that decision.

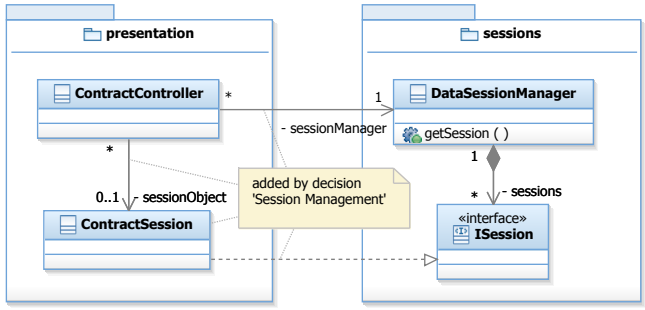


Figure 2.3: The *Server Session State* solution has been chosen and realized in the design model: *ContractSession* and three associations have been added.

There are further related follow-up decisions like session persistence [Kön09a], but they not required to understand the example in this thesis. The remainder of this chapter reflects on the example and discusses roles, development tools, and some characteristics of design decisions.

2.3 Decision Characteristics

Both decisions discussed previously are related to the design of the system, but decision templates like the one in Tab. 2.1 are generic and can be used for almost any kind of decision made in a project, not only those affecting the design. This thesis, however, concentrates on design decisions that imply changes in design models.

Two types of decisions have been presented here, one has direct and the other indirect impact on the design. Moreover, the decisions are related to each other,

one induces the other. Other relations are exclusion of other decisions, or, more general, compatibility relations.

The development of a software system involves a number of different stakeholders like the customer, a project lead, software architects, developers, testers, and others. The task of making decisions concerning the design, however, involves primarily two roles: first, depending on the importance of the decision, the decision maker(s), often an architect or modeler, and maybe others that are involved in making the decision. Second, the developers realizing the outcome of a decision in the design, typically modelers or programmers. These roles may also be taken by the same person. Some decisions, in contrast, might have a limited scope, for example, affecting one component only. Then a modeler might make and realize the decision. This is important because the different roles use different tools when making and realizing decisions.

The tool, or rather the strategy, for making decisions in this example is a simple decision template. There are tools available that support developers in preparing such decision templates and documenting decisions afterwards. Current research also concerns the setup of a knowledge repository as a collection of such decision templates.

2.4 Summary

This chapter gave an impression of the decision making task, which roles participate, and what consequences might occur. Important characteristics of making design decisions are the structured description and documentation of decisions, the consideration of alternative solutions, and the realization in the design. The second decision *Session Management* and its realization in the design model will be used throughout this thesis to illustrate new concepts.

Chapter 3

State of the Art

This chapter presents the state of the art in making and organizing design decisions, how they can be linked to design artifacts, and how model changes can be represented. The concepts of model changes are of interest because realizations of design decisions in model-based software development imply modifications in design models which can be described with model changes. A summary of the state of the art is also given in the respective parts of the remaining chapters of this thesis, if appropriate.

3.1 Design Decisions

Design decisions and, in particular, their definition and tool support for documenting and making design decisions are widely discussed in the literature. The state of the art of both aspects is presented here. A comparison with our conceptual solution is stated later in Sect. 6.8.

Design decisions play an important role in the community of software architecture knowledge management. Figure 3.1 sets design decisions in relation with other sources for architectural design knowledge like patterns, personal expertise, and requirements. Architectural knowledge is split into project-independent and project-specific parts in one dimension, and implicit (or tacit)

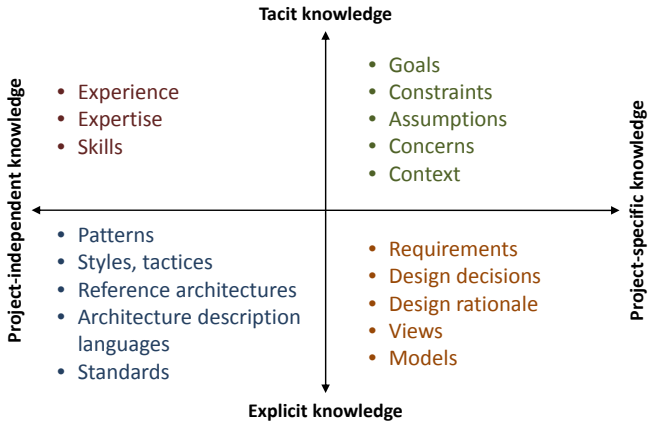


Figure 3.1: Architectural knowledge can be categorized in two dimensions, project-independent vs. project-specific and implicit (or tacit) vs. explicit knowledge; examples are given in each quadrant [BDLvV09].

and explicit knowledge in the other dimension. Tacit knowledge is undocumented knowledge that resides in the developers' mind, e.g. experiences or concerns in a project, whereas explicit knowledge is documented in one or the other way, e.g. pattern descriptions or design decisions. In this thesis, we focus on design decisions that are made in projects but also on reusable project-independent aspects of design decisions. The transition from tacit to explicit knowledge is also important but not in the scope of this thesis.

This section follows in large part the contents of *Software Architecture Knowledge Management* [BDLvV09], a book written by many members of the architecture knowledge community. It contains a summary of all relevant definitions and concepts of design decisions and available tool support.

3.1.1 Design Decision Definitions

One of the first appearance of design decisions was in the description of the Rational Unified Process [Kru03] where they are mentioned as important parts of a system's architecture. Later, templates have been introduced by Tyree and Akermann [TA05] which structure decisions according to important properties like issue, status, rationale, and assumptions. Other researchers suggested further properties for architectural design decisions: Kruchten's ontology for design decisions [Kru04] classified them and introduced relations between them

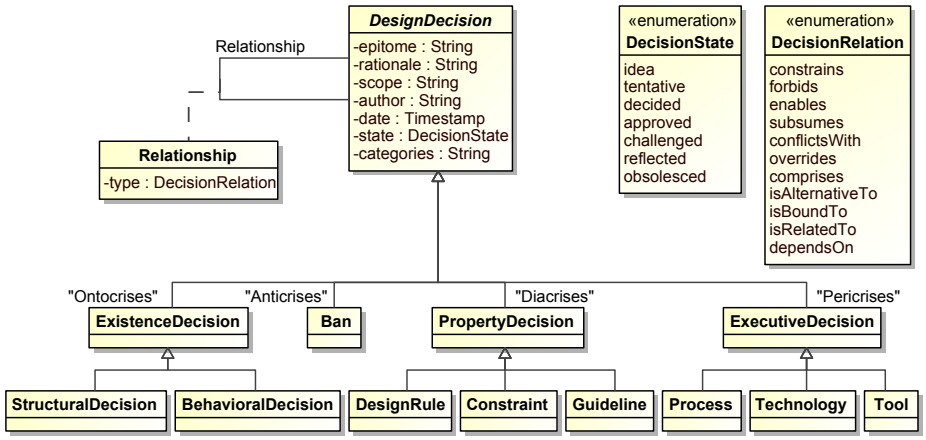


Figure 3.2: An overview of Kruchten's ontology for design decisions represented as a UML class diagram.

(see Fig. 3.2), Jansen et al. [JB05] defined them as first-class building blocks for the architectural design itself, and Zimmermann et al. [ZGK+07] focus their definition on reusability of architectural decisions. Moreover, the Core Model of architectural knowledge [dBFL+07] does not set the focus specifically on design decisions but integrates them as an important artifact into architectural knowledge in general.

The concepts in this thesis focus on reusable decisions, that is why we briefly discuss the two definitions that are closest to our intention. The first is Kruchten's ontology of design decisions, the second the SOAD (Service-Oriented Architecture Decision Modeling) metamodel by Zimmermann [Zim09].

The ontology depicted in Fig. 3.2 introduces four categories for decisions: *existence decisions* refer to concrete parts in the design; *non-existence decisions* or *bans* express parts or features that must not exist in the design; *property decisions* concern features about the design (not *in* the design); and *executive decisions* refer to the business environment, e.g. the development project and process. Each of them is further classified as shown in the figure. The decision attributes as well as the relationships between decisions are to a large extent self-explaining.

The SOAD metamodel, depicted in Fig. 3.3, defines two model types: the Reusable Architectural Decision Model (RADM) defines project-independent parts of design decisions, comprising the design entities *ADIssue* and *ADAlternative* as alternative solutions. The Architectural Decision Model (ADM) is a

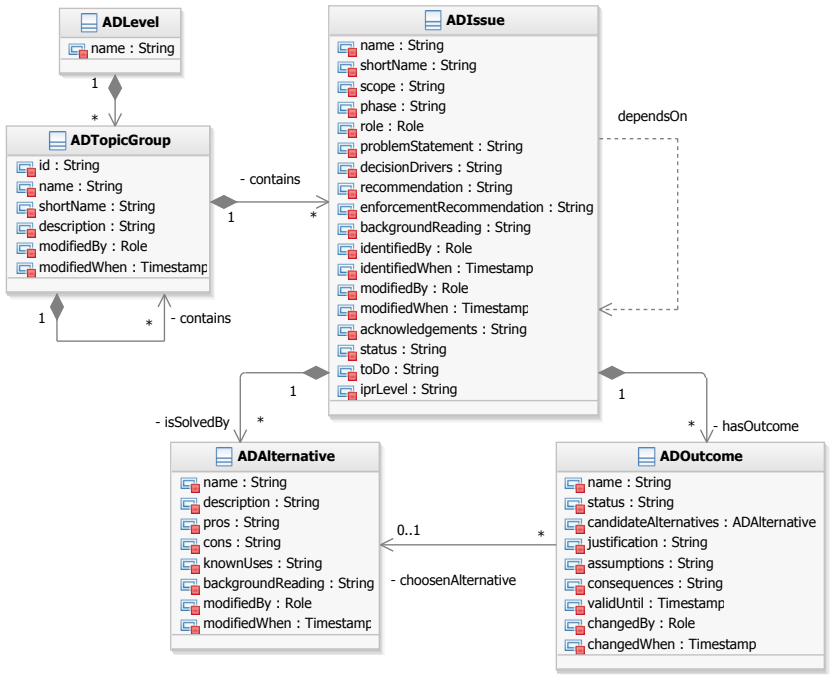


Figure 3.3: The SOAD metamodel for reusable architectural decisions by Zimmermann [Zim09]; unlike most others, this metamodel distinguishes between design issues, their alternative solutions, and the instances (outcomes) of decisions.

project-specific counterpart of the former and adds *ADOutcomes* which represent decision instances. This definition also includes a number of attributes that specify important properties for design decisions, and relations between them: *dependsOn*. The relations are discussed later in Sect. 6.7.

Other definitions of design decisions comprise similar attributes, but their focus is typically one particular aspect. For instance, Burge et al. [BB04] propose an approach to rationale knowledge management. Liang et al. [LJA10] concentrate on knowledge management and decisions in different tools and views on the architectural design. Tang et al. [TAJ+10] compare the latter and four other approaches with a focus on reasoning knowledge. Shahin et al. [SLK09] also compare several definitions and list their individual strengths. To conclude, there is a multitude of design decision definitions in the literature, but none of them involves design models.

3.1.2 Design Decision Tool Support

Most of the conceptual definitions in the literature are underpinned with a tool that implements the respective concepts. Just like their definitions, the tools focus on different aspects. This section gives a brief overview of existing tools and their capabilities, especially concerning their integration with design artifacts and how they support reuse of design decisions.

Many tools for decision management are web-based tools for collecting, documenting, and sharing architectural knowledge including design decisions: ADDSS (Architectural Design Decision Support System [CND07]), ADK Web Tool (Architectural Design Knowledge Web Tool [Zim09]), PAKME (Process-based Architecture Knowledge Management Environment [BWG05]), SEI wiki (a wiki developed by the Software Engineering Institute [BM05]). The Knowledge Architect [LJA09] is a repository-based tool suite for capturing, documenting, sharing, and managing architectural knowledge with focus on the integration with different tools, e.g. office tools. Archium [JvdVAH07] and SEURAT (Software Engineering Using RATionale system [BB08]) document design decisions and link them to the code of the system's architecture. AREL (Architecture Rationale and Elements Linkage [TJH07]) documents design decisions as profiled UML models for documentation and analysis purposes.

The tool presentation by Babar et al. [BDLvV09] distinguishes 23 use cases. The ones relevant for our concerns (which are discussed in detail in the subsequent Chapter 4) are listed below, along with a list of the aforementioned tools that support the respective use cases.

Integration with Design. Decision management systems are integrated with the design of a system if they maintain a link between design decisions and design artifacts. This corresponds to the use cases *UC3 – Trace AK*[1] and *UC10 – Integrate AK*.

AREL documents design decisions as UML models and, thus, they can be linked to design models. The links are used for traceability and design reasoning, but the decisions are neither project-independent nor reusable. Archium composes the architecture (source code) out of architectural decisions and SEURAT captures the rationale of a system and links it to the code. But both tools do not support design models. The other tools do not support these use cases.

[1] *AK* is an abbreviation for *architectural knowledge*.

Consistency. To our best knowledge, none of the existing approaches is able to automatically validate whether design models conform to made design decisions. This is to some extent similar to the use case *UC22 – Cleanup the architecture*.

Archium and SEURAT are at least able to check the consistency between made decisions and source code, but not design models. In addition, there are methods like ATAM (the Architecture Tradeoff Analysis Method [KKC00]) and SAAM (Software Architecture Analysis Method [KBAW94]) which require reviews and reports as quality assurance tools that check the consistency between the design and decisions or, generically speaking, design documentation. But this is typically manual work.

Decision Guidance. Most decision management systems support developers only in documenting and analyzing design decisions, but not in making decisions during forward engineering. Decision guidance corresponds to the use case *UC23 – Offer decision-making support*.

Archium is designed to create large parts of the architecture by making design decisions, also SEURAT supports developers in making decisions during the design process. But both do not support design models. The ADK Web Tool shows at least one case in which a transformation is triggered from the decision management system which induces updates in the design. Again, there are methods like RUP (Rational Unified Process [Kru03]), CMMI (the Capability Maturity Model Integration [Tea10]), or ATAM which enforce a decision making and documentation process but they do not provide tool support.

To summarize, there are many decision management systems available which support developers in documenting, capturing, and analyzing design decisions and to some extent also for guiding developers through the decision space. All tools have their own focus on one or a number of particular aspects. They can also be used to fulfill documentation and consistency requirements in development methods like ATAM, SAAM, and CMMI. However, none of them provides an integration with design models. One decision definition and tool support that is well suited for adding support for decision realizations in model-based software development is the ADK Web Tool because it supports forward engineering and has a separation of project-independent and project-specific parts.

3.2 Specification of Model Changes

Design decisions imply modifications in the design, including models in model-based software development. There are many different kinds of models like structural and behavioral models, and many different modeling languages. This section gives an overview of model types and existing approaches and technologies that are capable of capturing and describing modifications in models. A comparison with our conceptual solution is stated later in Sect. 5.10.

Structural models are typically used to describe static parts of the design. That could be the system architecture down to the internal structure of individual components. Behavioral models, on the other hand, describe the dynamics of software systems. A widely used modeling language is the UML [OMG10], a general purpose modeling language defined by the Object Management Group. It comprises different types of structural diagrams like class and component diagrams, and behavioral diagrams such as sequence and activity diagrams and state charts. Another modeling language is provided by MATLAB Simulink[2], a modeling, simulation, and analysis tool for dynamic and embedded systems. Examples for behavioral modeling languages are BPMN (Business Process Model and Notation [OMG09]) and Petri Nets. Both can, for instance, be used to describe business processes. The remainder of this section explains how changes in such models can be captured and specified.

3.2.1 Change Types

Whenever a model is changed, we want to assure that the resulting model is syntactically correct. The syntax of a model is defined by its metamodel, the definition of the respective modeling language. If a model conforms to its metamodel, it is syntactically correct. Models might also have particular semantics, which is in particular important for behavioral models. A modification which does not change the semantics of one model, might change the semantics in another model [Dij07]. However, in this thesis we only concentrate on structural models and neglect the semantics of dynamic aspects – that is future work.

Since models are defined in terms of their metamodels, descriptions of modifications in models must be defined in terms of the models' metamodels. We use UML class diagrams as examples in this thesis, as we have already seen in Sect. 2, but we want to support other modeling languages, too. Model modifications can also be described in terms of the meta metamodel of UML, which

[2]Available at: http://www.mathworks.com/products/simulink/

Type	Adding	Removing	Updating
Element level	Adding a class to a package or an event in a sequence chart	Deleting a parameter from an operation or an activity from an activity diagram	Moving an interface to another package or a property from one to another class
Attribute level	Adding a value to a multi-valued attribute	Deleting a value from a multi-valued attribute	Renaming a package or changing the visibility of a class
Reference level	Adding an inheritance relation between two classes	Deleting an inheritance relation between two classes	Changing the target of an association from one class to another

Table 3.1: Nine basic change types for MOF-based models; all modifications can be expressed by composing these basic change types; examples are given in the table.

is defined in MOF (Meta Object Facility [OMG06a]), a metadata management framework. Example modifications in UML models described in terms of MOF have already been shown in Fig. 1.1 on page 7. The basic change types for MOF-based modeling languages are listed in Tab. 3.1: changes are defined on model elements, their attributes, and on references between model elements [CRP07]. The modifications are either the addition of new parts or the removal or update of existing parts. Examples for each change type are given in the table.

There are different ways of how model changes can be specified. Two big areas exist in literature which deal with this question: model differencing and model transformations. Existing approaches and technologies are briefly discussed next. Later in Sect. 5.1, their suitability is discussed for specifying design decision realizations.

3.2.2 Model Differencing

One of the origins of model differencing approaches are version control systems. Analogously to differencing and merging source code, the same concepts are required for models [AP03]. That is, a local and a remote version of the same artifact can be compared and merged automatically or with user interaction. However, there is one big difference between text- and model-based comparisons: text is a linear sequence of characters organized in lines, whereas models conform to a particular metamodel. Although textual representations of models could be compared, for instance as XMI (XML Metadata Interchange [OMG07b]) or other text-based notations like EMF Text [HJK+09], there is an

important difference: text-based differencing approaches are not aware of models' metamodels and might produce syntactically incorrect models. This is why model-based differencing technologies are required.

Besides comparing and merging models in a version control system, another use case, which is of interest for automating realizations of design decisions, is the extraction of model changes from one model and their application to another model. For texts, this use case is often called *patching* and a set of textual changes is called *patch*. The research in model-based differencing is also going towards model patching [CRP07, Eys09] and beyond it, that is, model changes can even be customized and applied to arbitrary models [BLS+09]. This scenario implies an important problem: how to reference model elements that have been added, removed, or updated? Many approaches use *unique identifiers* for referencing model elements, but identifiers are not always available. Others use similarity heuristics or queries [KRPP09].

Next, we discuss several important characteristics of model differencing technologies. Tab. 3.2 lists existing approaches and their capabilities. The community of model differencing agrees on three parts for such a technology [BP08, KRPP09]: the *calculation* of model differences, their *representation* (abstract syntax), and their *visualization* (concrete syntax). The calculation and representation of model changes can either be *state-based* or *operational*. State-based model changes are calculated from two snapshots of a model and describe the state in the model before and after changes [FW07]. Operational model changes are recorded and describe a sequence of modifications [HK10].

Each of the existing approaches listed in Tab. 3.2 focuses on one or more particular aspects of model differencing. We only consider generic approaches; technologies for specific model types, like the difference calculation of business process models only [KGFE08], are not relevant here. The compared features are grouped in differencing capabilities, referencing strategies, and other features like support for *version control systems* and *patching models*. The feature *transferring changes* denotes the capability of applying changes to models that are different from the compared models from which the changes were calculated. *Merging changes* denotes the capability of automatically combining multiple similar changes into single generic changes (see Sect. 5.7.3 for details).

MPatch, our work, is also included in the table to compare it with existing approaches. EMF Compare [emf10] adds differencing and merging to version control systems in Eclipse. SiDiff [KWN05] is a generic differencing technology which operates on basic graphs and can be customized for specific modeling languages like UML. Cicchetti et al. [CRP07] present a metamodel independent approach to difference representation which also allows patching, i.e. the application of differences to other models. The ECL (Epsilon Comparison Lan-

Feature	EMF Compare	EPatch	AMOR	SiDiff	Cicchetti et al.	ECL/EML	MPatch
difference calculation	✔	✗[a)]	✗[a)]	✔	✗[a)]	✔	✗[a)]
difference representation	✔	✔	✔	✔	✔	✔	✔
difference visualization	✔	✗[b)]	✔	✔	✔	✗[b)]	○
id-based referencing	✔	✔	✗	✔	✗	✔	✔
similarity-based referencing	✔	✗	✗	✔	✗	✔	✗
query-based referencing	✗	✗	✔	?	✔	✔	✔
version control systems	✔	✗	✔	?	?	?	✗
patching models	✗	✔	✔	?	✔	✔	✔
transferring changes	✗	✗	✔	?	✗	✔	✔
merging changes	✗	✗	✗	✗	✗	○	✔

✔ – supported; ✗ – not supported; ○ – partially supported; ? – unknown

a) another technology is used for difference calculation

b) no dedicated graphical syntax but only textual syntax is available

Table 3.2: Overview of existing model differencing approaches.

guage [Kol09]) is a language specifically for matching models and EML (Epsilon Merging Language [KPP06a]) allows to merge models based on such matchings. Although AMOR [BLS+09] is an extension for version control systems, it includes a generic recorder for model changes, similar to what we require for specifying design decision realizations; however, this approach was not yet available at the start of our project. Otherwise it could have been adjusted to be used in our decision support tool.

Specialized technologies such as EWL (the Epsilon Wizard Language [KPRP07]) and Refactory, a dedicated tool for role-based refactoring [RSA10], comprise different strategies to achieve a similar functionality. The former uses the Epsilon Object Language [KPP06b] to specify arbitrary parameterized model changes, the latter uses a proprietary specification language for refactorings on models.

3.2.3 Model Transformations

Model transformations can also be used to specify modifications on models. In a wider sense, model differencing is a specialized kind of model transformations.

However, model transformation technologies have specific characteristics which we briefly discuss in this section. Most of them are from Mens and Gorp [MG06] and Czarnecki and Helsen [CH06].

In general, model transformations are either *model-to-text* or *model-to-model* transformations; the former produces text from a model, and the latter creates a new or modifies an existing model. Transformations that operate on only one model, which is modified, are called in-place transformations; otherwise the transformation creates or updates a target model from a source model. In that case, transformations may be *endogenous* (source and target metamodels are the same) or *exogenous* (source and target metamodels differ). We are only interested in endogenous and in-place transformations and also further properties like uni- or bi-directionality and incremental transformations are not relevant for us. Moreover, transformations are either automated or allow user interaction to influence the transformation.

Transformation specifications are either declarative and typically defined as a set of transformation *rules* in a specific transformation language, or operational and programmed in an interactive transformation language, possibly also structured in rules. Examples for declarative transformation languages are TGG (Triple Graph Grammars [SK08]), QVT Relations (Query/View/Transformation [OMG07a]), Story Diagrams [FNTZ00], and AGG (Attributed Graph Grammars [HET08]). Operational ones are QVT Operational Mappings [OMG07a] and ATL (ATL Transformation Language [Atl08]).

For the specification and use of design decision realizations, it must be possible to derive transformation specifications from example models (supported by TGGs by Wagner [Wag08], for instance) because we cannot expect a developer, who wants support for design decisions, to learn a transformation language. To our best knowledge, however, no existing transformation technology fulfills all of these requirements. This discussion is continued in Sect. 5.1.

3.3 Traceability

The realization of design decisions implies changes in the software design. In order to ensure that the design conforms to made decisions, we need to keep track of all affected model elements. In other words, we need to *trace* the affected model elements to the corresponding decisions. Some tools like Archium and SEURAT do this on the code level, but they do not support models. This section gives a brief excursion on traceability and outlines the state of the art for creating and maintaining traceability relations.

Two important purposes of traceability links are the support of developers during software maintenance and evolution by providing the corresponding documentation, as well as a compliance analysis between the software and the documentation. The evolution of software, however, requires to update the traceability relations accordingly. Mäder [MCH10], for instance, provides a methodology for updating the traceability relations semi-automatically; it records the changes made to UML design artifacts and uses predefined rules for keeping the requirements and the design consistent with each other.

The multitude of involved software artifacts (requirement documents, design artifacts, code, etc.), the stakeholders, and the high expectations the community has regarding traceability (impact analysis and integration of changes, coverage, verification and validation analysis, compliance analysis, consistent documentation, system inspection, etc.) make it hard to come up with an ultimate solution. An important part is the creation of traceability relations: *"The cost of identifying traceability relations manually clearly outweighs the expected benefits of traceability and makes organisations reluctant to enforce them, unless there is a regulatory reason for doing so"* [SZ05, p. 3]. For specialized purposes, however, such as an automated creation of traceability relations from requirement documents to design documents and code for documentation reasons, some (mostly commercial) tools like IBM Rational DOORS [IBM10] do a good job and are also used in large industrial projects. As a final remark, practical traceability support always depends on the CASE tools and their formats for storing documents and design artifacts, respectively.

A survey by Spanoudakis and Zisman [SZ05] concludes that most traceability approaches have some definitions and models for traceability relations, however, they mostly serve a special purpose (case-specific traceability links). Drivalos et al. [DKPF08] presented a metamodeling language for deriving case-specific traceability metamodels as a step towards a generic structure for traceability links. However, they focus on the definition and creation of such metamodels. Some common generic traceability relations emerged in several approaches like dependency relations, refinement relations, constraining relations, and documentation relations. The latter type refers to relations between design artifacts and their requirement documentation that is similar to the relation between design decisions and their affected design artifacts. In order to perform analyses on the relations, for example, to validate the conformance of a design against made decisions, the semantics of traceability relations must be clear and validation constraints must be defined. However, the validation is done manually today for design decisions even though there exist some proprietary approaches. This manual validation takes a lot of time. One of our goals is to reduce the manual effort and ideally provide an automated validation.

Chapter 4

Research Methodology

This research project analyzes support of design decisions in model-based software development and how it can be improved with respect to design models. In this chapter, we outline the research methodology and state our goals in more detail in three research questions.

1. Motivation. Literature studies, personal experience, and a workshop with software architects have shown that support for design decisions that are made in model-based software development is not satisfactory. Existing approaches for architectural design decisions do not operate properly on design models; we already discussed that in the previous chapter. The state of the art was analyzed and research directions were established in a workshop with a research staff member at IBM Research – Zürich in 2008. The results are elaborated in a technical report [Kön09a].

The identified problem is that both artifacts, design decisions and design models, are created and maintained in separate tools. State-of-the-art approaches do not include fine-grained links between made design decisions and individual elements within models. A fine-grained link between these artifacts could enable many useful use cases as stated next.

2. Goals. Design decision support in the literature involves numerous use cases, like the documentation, application, and validation of design decisions. Further use cases concern the management of design knowledge including design decisions, and its production and consumption. Our goal is to improve design decision support in model-based software development for such use cases:

1. Formerly separated design decisions and design models shall be linked to each other so that developers are able to navigate from documented design decisions to affected design model elements and vice versa. This shall improve the documentation of design decisions and design models.
2. Whenever recurring design decisions are realized in design models, these realizations may be recurring work. Using design templates as design decision realizations for design models, this recurring work shall be automated and, thus, easier and less error-prone. Developers shall be able to create such design templates with as little effort as possible.
3. After design decisions have been realized in design models, consistency between the documented design decisions and design models shall be validated. In case of inconsistencies, developers shall be notified and supported in restoring consistency between both artifacts.

3. Development. New concepts and a prototypic tool were developed that integrate design decisions and design models to realize all three aforementioned use cases. The artificial example from Sect. 2, which was initiated at the workshop in 2008, was used to develop and test the functionality. Furthermore, regression tests were used to continuously test the prototype whenever new functionality was added or third-party components were updated.

4. Validation. A proper evaluation for our concepts would comprise a feasibility study as well as an experiment in which the tool is applied in a running project. Since we did not do the latter experiment due to time constraints of this research project, we provide a set of correctness, applicability, and domain level tests. This is why we do not call it an evaluation but a validation of our concepts. The applicability tests cover refactorings and design patterns as design decisions, the domain level tests replay a realistic model of a real project to demonstrate the concepts on a real example. In addition, we received valuable feedback from experts in informal interviews about usability and success factors that are important for using the tool in practice. The interviews are, however, not meant to be an evaluation.

5. Conclusion. The validation shows that our concepts are applicable to realistic examples and that they are capable of automating recurring realization work of design decisions. The consistency checks come without extra effort. This shows the technical feasibility of the proposed goals and that they are applicable to realistic examples. However, an application under realistic conditions, e.g. under time pressure, and the effort for maintaining the binding could not be evaluated yet because the tool could not yet be applied in a running project. The latter as well as improvements on tool usability are future work.

The remainder of this chapter points out the three research questions in detail, each followed by a solution outline.

4.1 How to improve Design Decision Documentation?

Problem. Many decisions are made implicitly when designing a software system. Knowledge about and rationale behind the design are often not documented (tacit knowledge) but they reside only in developers' minds and vanish eventually. Tools for capturing such design knowledge exist and were already discussed in Sect. 3.1. They provide sophisticated metamodels, concepts, and tools for the analysis, for capturing, and for documenting design decisions. However, even if design decisions are documented, they are either captured and maintained separately from design artifacts or linked to source code. Concepts for linking documented design decisions to affected elements in design models do not yet exist.

Consequences. Since design decisions are documented separately from design models, laziness and oblivion cause imprecise documentation because decisions are not or too late documented. Thus, the quality and completeness of software design documentation suffers and the rationale behind the design is lost. This makes the design hard to understand for other developers, especially during system maintenance. Due to this bad comprehensibility, error fixing – which often has to take place under high time pressure – may introduce even more errors and decreases the comprehensibility further; this results in a vicious cycle. Projects may fail because of that reason [Par94].

Solution. One major cause of that problem is that the software is not documented properly. Therefore, we propose tool support for capturing design

decisions with only little overhead. That design knowledge is then available to other developers, following the intention of decision management systems. An integration of design decisions with design models links related documentation directly to the design artifacts. However, the two types of tools, modeling tools and decision management systems, shall remain separate because both may be used by different types of developers, namely software architects and modelers. Moreover, the concepts are independent of any concrete tool, which requires a technology-independent interface between the two types of tools.

Expected Benefits and Limitations. The expected benefit of an integration of design decisions and design models are threefold. First, in-place documentation of the design artifacts eases the understandability of the design. Second, storing design decisions as best practices spreads design knowledge to other developers. Third, instead of choosing their habitual solution, architects and modelers may consider other documented solutions by other developers which may be better suited. A remaining problem is that developers still have to decide which decisions to document and which not. If too many decisions are captured, developers run the risk of spending too much time producing design documentation that is probably never used. If too few decisions are captured, important parts of the design knowledge may remain undocumented. Developers must find the right balance which decisions to capture and which not. However, this discussion is out of scope for this thesis.

4.2 How to automate recurring Realizations of Design Decisions?

Problem. Whenever design decisions recur in the same or other projects, their realization in the design may be recurring work. Moreover, developers may have their own design templates like best practices which they use over and over again. This is all recurring manual work in similar contexts and manual realizations take a lot of time.

Consequences. Developers spend a lot of time realizing the same design fragments again and again. Of course, they gain experience in realizing such recurring design fragments and become faster each time they do it. But it is still manual and tedious work. Each time the developers may introduce errors and careless mistakes, especially because they are feeling confident in doing recurring work. This requires frequent and time-consuming design reviews.

Solution. Once a solution is realized for a particular decision, that solution can be captured for reuse. Then this proven solution can be used again every time that particular decision is made, and it may also be useful for other developers. One has to record the model changes and add them as a design template to the documentation of the corresponding design decision. The next time this decision is made (by the same or another developer), this design template can either be applied directly or it can be used as an inspiration for alternative realizations. A big challenge for this solution is to make the captured model changes applicable to arbitrary models.

Expected Benefits and Limitations. The major benefit for automating recurring realizations is to avoid tedious and error-prone modeling tasks. A side-effect is that developers, especially unexperienced ones, can learn from others by studying existing realizations. The drawback is that developers might use such design templates carelessly without putting it correctly into the context of their concrete model. Also, too many templates for a design decision might confuse developers and considering them all might be time-consuming. However, the last aspect is out of scope for this thesis.

4.3 How to ensure Consistency between Design Decisions and Design Models?

Problem. Let us assume that a developer changes a design model due to an explicitly made design decision. Other team members might also work on the model and make modifications. They may accidentally make changes that conflict with the decision of the first developer.

Consequences. An inconsistent design may lead to errors that are hard to find. In order to identify and fix such errors properly, a comprehension of the original decision is required. It might be impossible to gain that, in particular, if the decision is not documented and the responsible developer is not available, maybe because he or she left the team. If the design is not well understood, improper fixes may introduce further errors and, obviously, result in improper and incomplete design documentation. Hence, the design quality decreases.

Solution. Realizations of design decisions may imply changes of particular design model elements. We propose concepts to bind all affected model elements

to the respective design decisions, just like traceability links. Every invalid modification of these model elements is then reported to the developers. This tackles the problem at the source by warning developers that the design model does not comply anymore with made design decisions.

Expected Benefits and Limitations. The advantage is that inconsistencies between made design decisions and affected design models can be detected automatically. This way, potential errors can be detected immediately. In case inconsistencies are found, the documentation of the design decision should give information about which developers are or were working on the conflicting parts. Together, the developers can then discuss how to correct the inconsistencies. However, this assumes that developers document their design decisions properly. Moreover, the consistency check can be arbitrarily complex, especially if the semantics of models are taken into account.

The following chapter introduces concepts for specifying model changes with which recurring realization work for design decisions can be automated. Then, Chapter 6 presents novel concepts to elaborate the solutions discussed for all three research questions.

Chapter 5

Model-Independent Differences

Tool support exists for defining and applying refactorings[1], design pattern elaborations, design decisions, and other recurring changes on code. For models, there is also tool support available for specific use cases. However, an automated specification of model changes from example models such that they are applicable to arbitrary models is only partially supported. This chapter presents model-independent differences, a concept for the representation of model changes which can be created from example models and are applicable to arbitrary models of the same kind. Hence, model-independent differences can be used for defining and applying refactorings, design pattern elaborations, realizations of design decisions, or other recurring changes directly on models.

The running example of Sect. 2 was chosen for explaining design decisions but it is not well-suited for illustrating model changes. Therefore, another example is introduced which shows a refactoring. Two aspects will be shown which are not covered by the other example: first, model changes are captured from one model and applied to another model; second, model changes are generalized.

Figure 5.1 shows on the left-hand side the first UML model M_A of this example, a package *data* containing several classes, some of them owning an attribute

[1] Refactorings for models restructure a given model in order to improve the quality of the model without altering the functionality.

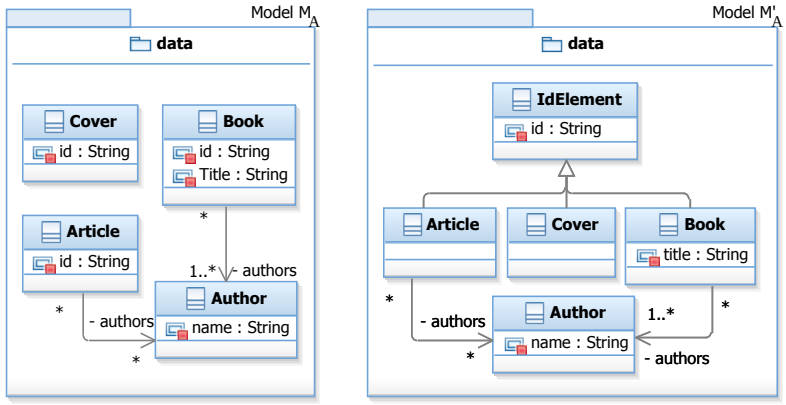


Figure 5.1: The refactoring *extract superclass* has been applied to model M_A on the left-hand side: several classes contain an attribute *id : String* which is extracted into a superclass *IdElement*; the resulting model M'_A is shown on the right-hand side.

id : String. The refactoring *extract superclass*, defined by Fowler [Fow99], was applied to it and the resulting model M'_A is shown on the right-hand side in the figure. A new class *IdElement* has been added including the attribute *id : String* which has been removed from the other three classes; instead, a generalization has been added between them and the newly added class. In addition, the attribute *Title : String* of the class *Book* has been renamed to *title*. The individual model changes are summarized in Tab. 5.1.

#	Change type	Change description
1	add class	*IdElement* to *data*, including attribute *id : String*
2	delete attribute	*id : String* from *Book* in *data*
3	delete attribute	*id : String* from *Cover* in *data*
4	delete attribute	*id : String* from *Article* in *data*
5	add generalization	from *Book* in *data* to *IdElement*
6	add generalization	from *Cover* in *data* to *IdElement*
7	add generalization	from *Article* in *data* to *IdElement*
8	update attribute	*Title : String* in *Book* to *title*

Table 5.1: List of changes (Δ) in model M_A (Fig. 5.1), grouped by change types.

Suppose the user wants to apply the same refactoring to another model shown in Fig. 5.2. This UML model M_B contains a package *customerdata* including three classes *Customer*, *Contract*, and *Invoice*, and several associations between them. Since we extracted the refactoring already from model M_A, it would save the user time and effort to automatically apply the same intentional changes (#1–7

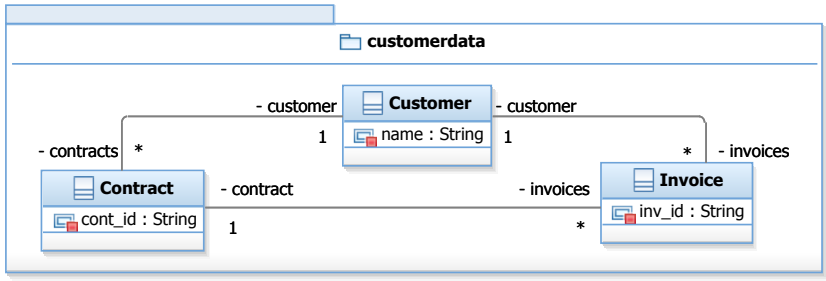


Figure 5.2: Another model M_B to which the refactoring *extract superclass* should be applied.

in Tab. 5.1) also to model M_B. This is, however, not possible with existing concepts due to the following reasons:

- The package to which the changes should be applied is named *customerdata* and not *data*.
- The classes from which a superclass should be extracted (*Invoice* and *Contract*) are named differently and their number differs (2 instead of 3).
- The attributes that should be deleted are named differently (*inv_id* and *cont_id*) and their number differs (two instead of three).

Existing model differencing technologies are either not capable of transferring model changes from one model to another, or they fail as soon as the other model, model M_B in this case, differs from the original model M_A – these differences have just been listed. The idea of model-independent differences is to uncouple model changes from the original model in order to make them applicable to other models. The latter is achieved by generalizing model changes. As we will se later, the resulting model changes are in fact applicable to other models like model M_B and changes do not need to be made manually anymore. Specialized approaches like EWL, in contrast, are capable of applying such changes, but their specification must be done manually in a specific language.

This chapter is structured as follows. The requirements for model-independent differences are discussed in Sect. 5.1, then Sect. 5.2 introduces the process of using them, and the subsequent Sect. 5.3–5.6 define model-independent differences in detail. Afterwards, sections 5.7 and 5.8 explain the generalization and application of differences, the key features for a differencing technology to be used for describing model changes for design decisions. Sect. 5.9 introduces a binding, Sect. 5.10 discusses related work, and Sect. 5.11 concludes this chapter.

5.1 Requirements for describing Model Changes

This section discusses the requirements for a technology that is capable of describing model changes and specifying realizations of design decision outcomes. There are in particular three use cases that must be supported:

(A) specifying model changes from examples,
(B) applying model changes to other models, and
(C) checking that all applied changes prevail in a model.

There are two potential areas of research which are capable of describing changes in models: model transformations and model differencing. Both of them were already introduced in Sect. 3. Here we compare their suitability concerning the three use cases (A)–(C). The section concludes with the justification of the decision to use a model differencing instead of a model transformation technology.

A. Specification of Model Changes by Example

Whenever a design decision including possible solutions is documented, the description of solutions might contain example realizations in the form of sample code, model fragments, textual descriptions, etc. In model-based software development, the obvious way is to provide example models along with a description of the solutions. Our new idea is to provide realization specifications as sets of model changes that can automatically be applied to any model instead of only describing examples. This reduces the error rate when re-modeling the solution in a design because the solution is applied as-specified, and it saves time because the manual modeling task is in large part obsolete.

A specification of model changes from scratch, as it is often the case for model transformation rules, is not feasible for two reasons: first, we cannot assume that every developer is familiar with a model transformation language; second, it takes a lot of time to specify new transformation rules. Hence, a specification of model changes by-example is more appropriate because the modeling language of the models is used. Also, the specification of model changes from example models is fast because the examples are either already available in a model or the developer can quickly create an example model.

This requirement applies to all model differencing technologies that are capable of difference calculation [TBWK07, BP08, XS05, OWK03, CRP07]. Few model transformation technologies also include concepts for deriving transformation rules from example models [Var06, Wag08].

B. Application of Model Changes to other Models

A set of model changes should be applicable to arbitrary other models. Since the application domain of design decisions does not often change, we assume that the models, to which model changes should be applied, are of the same kind as the model from which the changes were created. For example, if model changes were specified from UML models, it is sufficient that they are again only applicable to UML models.

Although model changes could be specified by example models, they should be applicable to other models whose contents differ from the example models. This is of course not always possible, some changes require the existence of particular parts in the model which we call context. Model changes shall be applicable as soon as their context exists in the target model. Hence, the computed model changes and their context description shall be *generalized* in a way that they are applicable to a wide range of models. Most of the studied model transformation technologies support this, but hardly any model differencing approach does.

The application of model changes must be interactive and comprehensible for developers. Whenever model changes are applied to a model, developers shall have control over the process: all affected model elements shall be shown and the developer shall be able to manually specify to which model elements the changes will be applied. This feature is not supported by any technology so far.

C. Consistency between Model and Applied Changes

Whenever model changes are applied to a model, we want to assure that these changes prevail; in other words, we want to assure that the model reflects all applied changes even when it evolves over time. This requires to keep track of all changed model elements and to check whether modifications in the model might violate any applied change. This consistency check can either be performed 'live' whenever developers are working with the model, or the check can be 'offline' and performed on demand. Developers committing the model to the repository, simply saving the model, or making design decisions are example triggers for consistency checks on demand. Either of these solutions shall be integrated.

Some transformation technologies offer traces or correspondence links between the transformed models, but rather for incremental transformations than for consistency checks. Nevertheless, these links could be exploited for consistency checks. Model differencing technologies do not have such a feature.

Comparison

Here we compare both technologies for model differencing and model transformations, and make the decision which one to use. Table 5.2 lists the evaluation at a glance. There is even another option which was neglected so far, namely the development of a new technology from scratch – but this alternative has been discarded because of the high effort and developing time required.

Requirement	Model Transformations	Model Differencing
A. Specification by example	With few exceptions, transformation specifications (typically rules) must be specified by the user.	The nature of model differencing is to extract changes from models, which is already by-example.
B. Applicability to other models	The nature of model transformations is to apply transformations to arbitrary models; however, the transformation is usually automated, i.e. not interactive.	Some model differencing technologies support 'patching'; that is, applying changes to other models. There are, however, many limitations. The application is usually interactive.
C. Consistency between model and applied changes	Some model transformation technologies produce trace artifacts as a link between models to enable incremental transformations.	There is usually no way of validating whether changes in the model conform to any previously applied changes.

Table 5.2: Comparison of the requirements of the technologies *Model Transformations* and *Model Differencing* for specifying realizations of design decisions.

Neither of the technologies meet all requirements. Although model transformation technologies come with a general applicability to arbitrary models, their lack of interactive transformations and the required knowledge of a specific transformation specification language are strong arguments against them. Model differencing technologies, on the other hand, are already by-example and provide an interactive application of changes, but the tradeoff is the limited applicability to other models and lacking traceability links.

Based on this analysis, we decided to build on top of existing model differencing technologies and improve on flexibility and applicability to other models. The following section outlines a new process which integrates with existing technologies and which meets requirements A and B. Then we present a new metamodel for model changes with focus on a general applicability to models. Requirement C is dealt with later in Chapter 6.

5.2 Process for Creating and Applying Model Differences

The creation and application of model-independent differences can be divided into six steps that are introduced in this section to give an overview of how the technology is used. The subsequent sections define the concepts and describe each of the six steps in detail.

The name, *model-independent differences*, reflects two aspects. First, the new format for describing model changes is self-contained and does not depend on or use any model. Second, and in contrast to traditional model differencing concepts, model-independent differences are not restrained to the original models from which they were calculated but they are applicable to arbitrary other models of the same type. The key feature is so-called *symbolic references* which are explained in detail in Sect. 5.4.

Several terms are used in this chapter whose definition is required for properly understanding the concepts. *Model changes* are descriptions of modifications in a model; examples are adding or deleting elements (more examples in Tab. 3.1 on page 24). The term *model differences* refers to a set of model changes that can be calculated from two versions of a model; they describe how the two model versions differ. An example is the set of all changes in Tab. 5.1. The term *delta* (Δ) is often used in the literature to represent a concrete artifact containing model differences; we also use the symbol Δ to represent model differences.

The overall process is divided into six steps as sketched in Fig. 5.3. The first part involving steps 1–3 covers the creation of model-independent differences, the second part (steps 4–6) covers their application. Model M_A, from which the model changes are calculated, is available in two versions, an unchanged original M_A and a changed version M'_A. Figure 5.4 illustrates the process for change #1 of the example in Tab. 5.1, adding a new class *IdElement*.

1. Calculation of model differences: $\Delta = \delta(M_A, M'_A)$
There are many sophisticated approaches for the calculation of differences between two versions of a given model M_A (cf. Sect. 3.2). Therefore, we make use of existing difference calculation algorithms that are configurable and extendable instead of reinventing the wheel. A difference calculation algorithm δ calculates the differences Δ from two models M_A and M'_A (or three models in case of 3-way comparison [FW07]). Most technologies, however, produce deltas that refer to and require the original model M_A – that is meant literally, i.e. the delta Δ is invalid without the original models. This is why we call such a delta *model-dependent differences*.

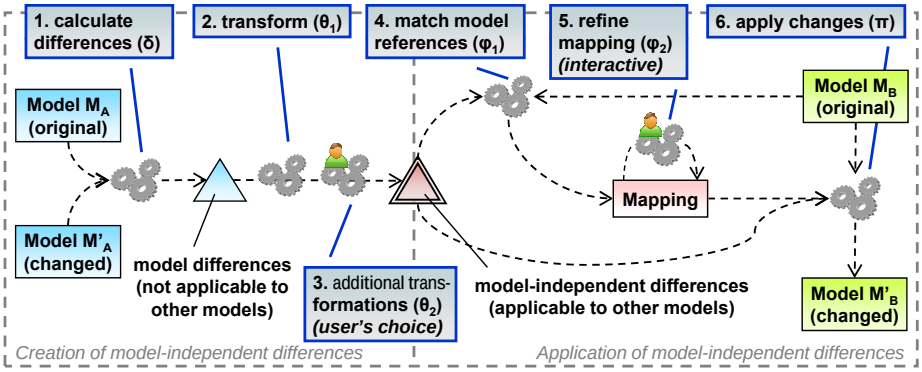


Figure 5.3: The overall process for creating model-independent differences from a model M_A (steps 1–3) and applying them to a model M_B (steps 4–6); steps marked with (👤) require user interaction.

The example in Fig. 5.4 simplifies the change in the model-dependent differences as *"Add 'IdElement' to 'data' in 'M_A'"*.

2. Transformation into model-independent format: $\Delta_a = \theta_1(\Delta, M_A, M'_A)$
The model-independent format, created by a transformation θ_1, decouples all changes from the original model M_A and the resulting model-independent differences are in particular applicable to other models. The key for that feature is the concept of *symbolic references* in Δ_a which replace direct references to elements of the original model M_A in Δ. The framework supports different types of symbolic references, see Sect. 5.4 for details.

The example in Fig. 5.4 simplifies the change in the model-independent differences as *"Add 'IdElement' to 'data'"*. This change is, in particular, not bound anymore to the original model M_A.

3. Generalization of atomic model changes: $\Delta_s = \theta_2(\Delta_a)$
This step applies additional transformations θ_2 to changes of the initially created model-independent differences; each of these additional and optional transformation yields a particular purpose, for instance, to make changes applicable to multiple model elements, or to combine similar changes. The idea of generalizations is to keep the intention of the original changes but to extend their scope of application; that is, making them applicable to other models, too.

The example in Fig. 5.4 simplifies the generalized change as *"Add 'IdElement' to '*data*'"*. The *'*'* in the model reference *'*data*'* denotes the generalization and can be read as wildcards here.

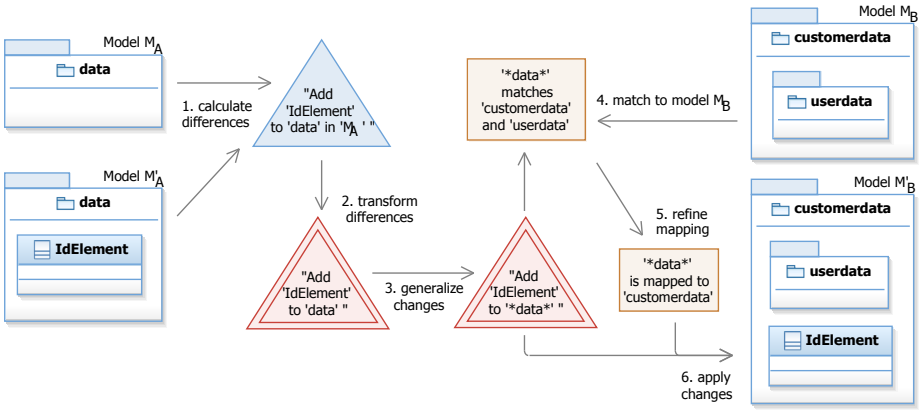


Figure 5.4: A simplified illustration of the process from Fig. 5.3 applied to change #1 of Tab. 5.1 and simplified versions of model M_A and M_B.

4. Initial (automated) matching of changes to model elements of an arbitrary model: $\Psi = \psi_1(\Delta_s, M_B)$
Each change in model-independent differences specifies individually via symbolic references to which model elements it is applicable. These symbolic references are responsible for identifying that set of elements in an arbitrary model; this step is typically called *matching* in the literature. Doing that for all changes yields an initial mapping Ψ from changes to model elements.

The example in Fig. 5.4 matches both packages *customerdata* and *userdata* for the model reference *'*data*'*.

5. Refinement of the mapping: $\Psi' = \psi_2(\Psi)$
The user may review and refine the initial mapping Ψ. This is required whenever a change is mapped to too many or too few model elements or when changes are not applicable to the set of matched model elements. It is in particular also possible to map a single change to multiple model elements; an example is the movement of several elements (e.g. UML classes) from one container (e.g. UML package) to another. The user may also, for each change, freely add/remove model elements to/from the mapping – as long as the change is still applicable to all selected model elements.

In the example in Fig. 5.4, the initial mapping was reduced to the package *customerdata*.

6. Application of changes: $M'_B = \pi(\Delta_s, \Psi', M_B)$
In this step, the framework automatically modifies all mapped model elements in model M_B according to the changes in Δ_s. This step is often called *merging* in the literature. Conceptually, the application of changes is straightforward except for some minor obstacles that are discussed in Sect. 5.8.

The example in Fig. 5.4 shows the resulting model M'_B after the change has been applied to the model element *customerdata* as specified in the mapping.

Conclusion and Discussion

The process of using model-independent differences consists of six steps, each of which is explained in detail in the subsequent sections. The actual calculation of model differences is performed by other technologies, the concepts presented here build on top of that.

In contrast to related work, our proposed process supports a generalization of model changes in form of optional transformations. Furthermore, the mapping Ψ, that specifies which model elements will be changed, can not only be reviewed, as is the case in most related work, but also refined. Model changes can then, in particular, be applied multiple times. The corresponding sections explain these features in detail.

5.3 The Metamodel for Model-Independent Differences

The important characteristics of model-independent differences are self-containment and applicability to arbitrary models. This and the following two sections introduce concepts in terms of a the metamodel for model-independent differences that enable these characteristics. The core of the metamodel is covered in this section, additional concepts for symbolic references and model descriptors are covered in Sect. 5.4 and 5.5, respectively.

This section starts with several decisions we made concerning the design of model-independent differences. Based on the outcome of these decisions, the metamodel is presented and explained. The section concludes with a brief discussion of the presented concepts.

5.3.1 Design Decisions

Based on experiences, literature studies, and related work, we made four fundamental design decisions concerning the definition of model-independent differences: the scope of model differences, the applicability to other models, the internal representation of model differences, and the set of supported change types. Understanding these decisions and the rationale behind them helps to understand the metamodel. The first decision concerning the scope of model-independent differences is discussed in the following table.

Decision 1:	**Scope of model-independent differences**
Issue:	Related work differentiates between the calculation, representation, visualization, and application (also known as merging) of model differences [BP08]. All of these aspects are required for a model differencing technology.
Alternative 1:	Development of a complete technology that realizes all of the mentioned aspects.
Pros:	+ Independence of third party tools. + Perfect adjustments to our needs.
Cons:	− High effort for development and maintenance. − Realization of already existing concepts is not reasonable for a research project.
Alternative 2:	Use existing tools for tasks that are out of focus.
Pros:	+ Saves work and time. + Collaboration with community of other tools.
Cons:	− Dependency on other tools and their development. − Effort to get familiar with the other tools' concepts.
Outcome:	Reuse as many components as possible (Alternative 2).
Assump.:	Several other tools have a public API and can easily be reused (technically and legally).
Just.:	Reuse existing and sophisticated open source solutions for the calculation and visualization of model differences and concentrate on new concepts for their representation and application.

The first decision is to build model-independent differences on top of existing difference calculation concepts. The next decision concerns which types of models will be supported by model-independent differences.

Decision 2:	**Applicability to models**
Issue:	Although this thesis is focusing on the UML, other modeling languages may be supported, too. That implies the need for supporting different metamodels because model changes are defined in terms of the model's metamodel.

Alternative 1:	Optimize metamodel for a particular set of modeling languages (SiDiff [TBWK07] is an example for that strategy).
Pros:	+ Fast and precise detection and application of model changes. + Notation-specific details can be addressed.
Cons:	− Support for other types of models requires high configuration effort.
Alternative 2:	Define model differences generically in terms of a meta metamodel like MOF [OMG06a]. Reflection could be used to generically operate on the metamodel of the used modeling language.
Pros:	+ Any MOF-based modeling language is supported.
Cons:	− Only changes that can be defined in terms of the meta metamodel can be expressed, metamodel specific changes are not possible. − The algorithms for creating and applying model changes might be slower because of the use of reflection.
Outcome:	Generic specification of model differences based on MOF (Alternative 2).
Assump.:	All modeling languages of interest are based on MOF.
Just.:	MOF is widely used by many modeling languages. Supporting many modeling languages is of higher interest than supporting only a single one.

As outcome of the second decision, multiple types of models will be supported, e.g. the UML, SysML, BPMN, or any other MOF-based modeling language. Reflection, a property of MOF-based metamodels, will be used to access and modify models. The next decision discusses the representation of model changes.

Decision 3:	**Internal difference representation**
Issue:	Two concepts dominate in the literature to represent model differences: declarative and operational. Both coming with their specific advantages and disadvantages.
Alternative 1:	Each model change is represented declaratively by the state of the model before and after the actual change. This kind of representation typically results from state-based model comparisons.
Pros:	+ Flexibility: by interpreting the declarative change description, the change can be re-applied and undone. + It is irrelevant who or how changes have been made, e.g. manually or by automated model processing.
Cons:	− Only changes reflected in the resulting model are covered, intermediate changes are lost.[2]
Alternative 2:	All changes are recorded as they are made by the user.
Pros:	+ The changes are described as precisely as possible [HK10].
Cons:	− Changes must always be recorded while the model is edited. − Only editors may be used that support recording of model changes.

[2]Example: a *move* and *update* could be detected as a *delete* and *add*.

Outcome:	State-based model change descriptions (Alternative 1).
Assump.:	The versions before and after model changes are available.
Just.:	The user shall not be limited to particular modeling tools. So it must be possible to compare two snapshots of a model. A hybrid approach could be used in the future to increase precision.

The outcome of this decision induces a state-based and declarative representation of model changes. A hybrid approach including a change recorder could be added in the future. The last decision concerns the types of changes which will be supported by model-independent differences.

Decision 4:	**Supported set of change types**
Issue:	All changes that can possibly be made in a model are described in terms of its metamodel. However, the basic operations *add* and *delete* can be used to describe any change. Additional change types like movements and updates increase the accuracy of change descriptions. A sub-issue is the representation of complex changes.
Alternative 1:	Support only additions and deletions.
Pros:	+ The change metamodel is simple. + Tooling implementation is easy.
Cons:	− Low accuracy for describing changes; the user may not recognize his or her actual changes in the calculated set of model changes.
Alternative 2:	All change types that are possible for a particular metamodel.
Pros:	+ Covering all possible change types of the given metamodel provides the best accuracy for describing model changes.
Cons:	− The change metamodel and tooling realizations might get very complex, depending on the model's metamodel.
Alternative 3:	A subset of all possible change types.
Pros/Cons:	± A compromise of the previous two alternatives.
Outcome:	Support most common change types (Alternative 3).
Just.:	The concepts shall support most common change types: addition, deletion, and update of model elements, their attributes, and references between them [BP08]. The metamodel for model-independent differences shall also be extendable for other change types.
Conseq.:	Other change types shall be documented in model-independent differences even though they are not supported. Complex changes are described by composing basic changes.

As a result of the last decision, the supported change types covers the following basic types: addition, deletion, and update of model elements, references, and attributes. However, the concepts are extendable for further specific types.

5.3.2 The Metamodel

The concepts of model-independent differences are defined in terms of a metamodel which is presented in the remainder of this section. It comprises three packages, a core package called *indepdiff* (an abbreviation of 'model-independent differences') and two packages *symrefs* and *descriptors*. The latter define how model elements are referenced and stored, respectively. The dependency relations between all three packages are shown in Fig. 5.5.

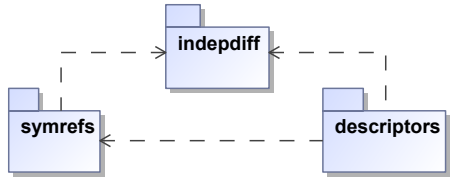


Figure 5.5: Dependencies between the different packages of the metamodel for model-independent differences.

The Package *indepdiff*

The package *indepdiff* describes eleven change types in total, listed in Tab. 5.7: nine concrete change types, one for composed changes, and one for unsupported changes. Six of them are presented in detail, the others are defined analogously.

	Element	**Reference**	**Attribute**
Add	*Adding* a new model *element* *	*Adding* a model element to a multi-valued *reference* *	*Adding* a value to a multi-valued *attribute*
Remove	*Removing* an existing model *element*	*Removing* a model element from a multi-valued *reference* *	*Removing* a value from a multi-valued *attribute*
Update	*Moving* an existing model *element* to another parent	*Updating* the referenced model element of a single-valued *reference*	*Updating* the value of a single-valued *attribute* *
Change group *		*Unknown change* *	

Table 5.7: An overview of all change types supported by model-independent differences; changes marked with (*) are explained in detail.

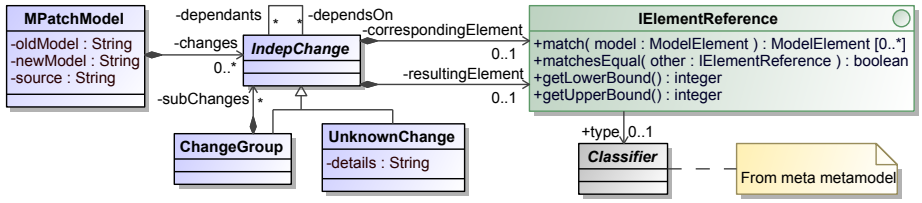


Figure 5.6: An excerpt of the metamodel for model-independent differences showing the container *MPatchModel*, the abstract change type *IndepChange*, and two concrete change types for groups and unknown changes.

The excerpt in Fig. 5.6 shows the abstract superclass *IndepChange* for all changes along with the two change types for change groups (*ChangeGroup*) and unknown changes (*UnknownChange*); the complete package is depicted in Appendix A. *MPatchModel* represents model differences between two models for which the attributes *oldModel* and *newModel* may be used to store additional information. *Source* may be used to store information about the source from which the model-independent differences were created, like the difference calculation algorithm.

The nine atomic change types are added, removed, or updated model elements, their attributes, or references. Other change types, for instance, order changes of an ordered reference, are currently not supported. Whenever the creation of model-independent differences (discussed later in Sect. 5.6) detects such an unsupported change type, an *UnknownChange* will be created including appropriate information about the original change in the property *details*. To implement unknown changes, a new change type can be added to the metamodel by adding a new subclass of *IndepChange* with respective properties to describe the desired change.

Change groups do not describe any concrete change in a model but they may contain other changes. This way, complex changes can be composed of several other changes, and the changes may be hierarchically structured – this is an instance of the composite design pattern [GHJV95].

References to model elements. Each of the atomic change types describes a particular change on one or more model elements or their properties. The property *correspondingElement*, common to all changes (cf. *IndepChange* in Fig. 5.6), specifies that particular set of model elements in the unchanged state, and *resultingElement* in the changed state. They contain instances of *IElementReference*, an abstract concept which we call *symbolic references* that *match*es

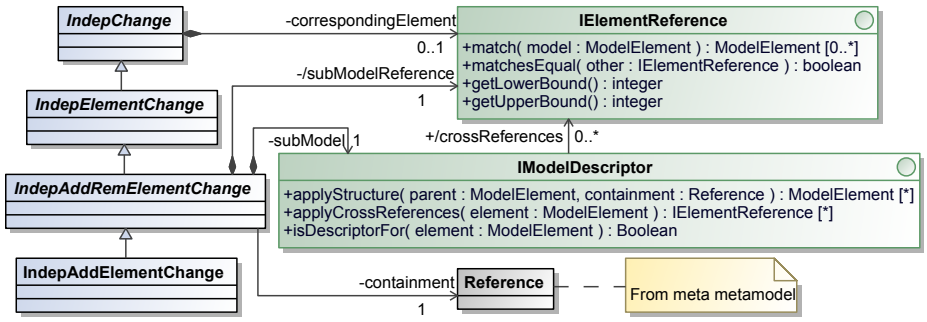


Figure 5.7: The metamodel for changes describing added elements; implementations of *IModelDescriptor* contain a self-contained description of the element that was added.

such a set of model elements in a given *model*. *LowerBound* and *upperBound* are used to constrain the size of that set. Details about symbolic references are explained in the subsequent section, for now it is sufficient to understand them as a concept for referencing arbitrary model elements.

Change #2 in Tab. 5.1 on page 36, for example, describes a change on the model element *Book*. Hence, a symbolic reference describing the corresponding element for this change would contain some information to resolve the class *Book* in model M_A (cf. Fig. 5.1 on page 36). It is the responsibility of the symbolic reference, *how* this (in this case single-valued) set of model elements is determined.

Changes on model elements. Model changes on elements comprise the addition, deletion, and move of model elements. A model element is a node in the graph-representation of a model. These change types do not include modifications of model element properties (attributes of graph nodes) or references between them (graph edges). In the metamodel excerpt in Fig. 5.7, the addition in presented in detail; a deletion is represented similarly, a move slightly different without the need for a model descriptor. The following information is required to describe an addition: *what* is added and *where*.

Where: The symbolic reference of *correspondingElement* defines the parent to which the new element is added. The *containment* defines the property of the parent to which the new element is added; the containment property is of type *Reference* which is part of the MOF meta metamodel. Instances of

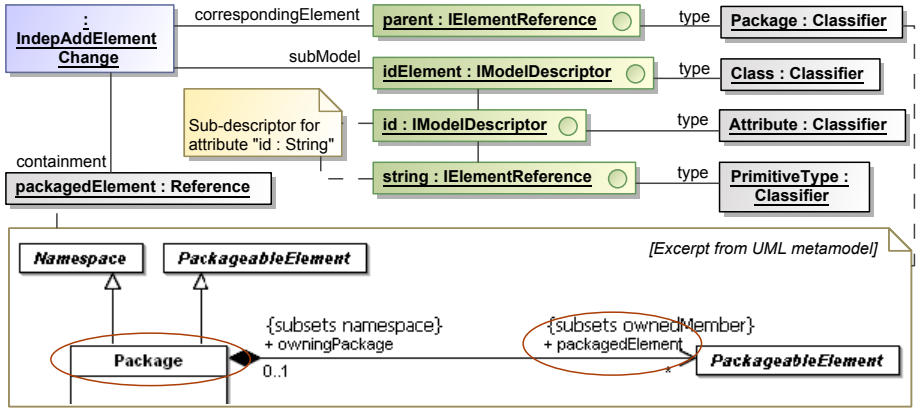


Figure 5.8: An instance of an *IndepAddElementChange* as a UML object diagram; the containment reference and the types are parts of the UML metamodel.

Reference are, amongst others, containment relations in the UML metamodel. The following example clarifies this containment reference.

What: The model descriptor of *subModel* is a concept for describing arbitrary model elements; it is a self-contained description of the new element.

Change #1 in Tab. 5.1 is an example for an added element, a UML class to be precise. An instance of *IndepAddElementChange* describes such a change, as illustrated in an object diagram in Fig. 5.8. The corresponding element (parent) is a *Package*, defined in the UML metamodel. The parent's property which holds the added class is *packagedElement*, a *Reference* in the UML metamodel. The new element *IdElement* is described with a model descriptor which may also contain sub model descriptors such as the one for the attribute *id : String*. The type of the attribute (*String*) is, however, part of a basic UML profile and, hence, a *cross reference* to the type that is contained in that profile. *Cross references* of model descriptors are references from the described model elements to other model elements that are not described by this model descriptor. Cross references are again stored as symbolic references.

Changes of model element references. Model changes on references comprise the addition, deletion, and update of references between model elements. A reference is a directed edge in the graph-representation of a model, for instance, an inheritance relation in UML models. Here we only present the addition and deletion in detail (cf. metamodel in Fig. 5.9); an update of a reference is

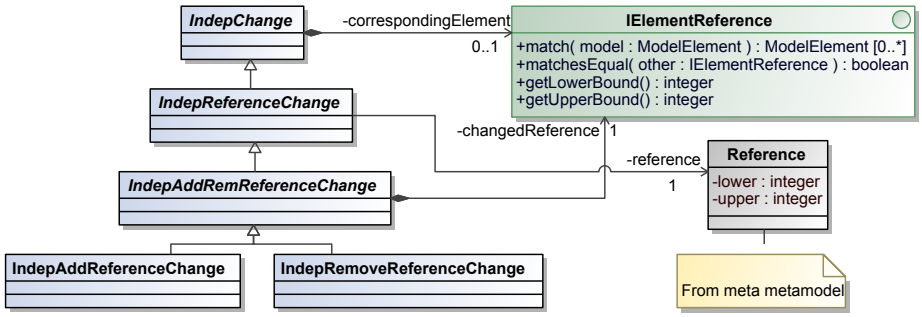


Figure 5.9: The metamodel for changes describing added and deleted references.

described slightly different with two instead of one reference targets. Again, the following information is required to describe an addition or a deletion of a reference: the *source* or owner and the *target* of the added/deleted reference.

Source: The symbolic reference of *correspondingElement* defines the source/owner of the added/deleted reference. The *reference* specifies the property of the source which holds the reference to be added/deleted.

Target: The symbolic reference of *changedReference* defines the target of the added/deleted reference.

Change #5 in Tab. 5.1 is an example for an added reference, a UML generalization from the class *Book* to the class *IdElement*. An *IndepAddReferenceChange* describes such a change. The source is described by the *correspondingElement* (the class *Book*), the target by the *changedReference* (the class *IdElement*).

The distinction between added/deleted and updated references is defined by the cardinality of the *Reference* of the model's metamodel (*lower* and *upper*). If $upper > 1$, as it is the case for UML inheritance relations, then references are added or deleted; if $upper = 1$, as it is the case for type reference of UML attributes, for instance, then references are updated.

Changes of model element attributes. Model changes on attributes comprise the addition, deletion, and update of attributes owned by model elements.[3] Examples are the attributes *name : String* and *isAbstract : boolean* of

[3] UML attributes (cf. changes #2–4 in Tab. 5.1), on the other hand, are just model elements in contrast to meta attributes of model elements to which we refer here.

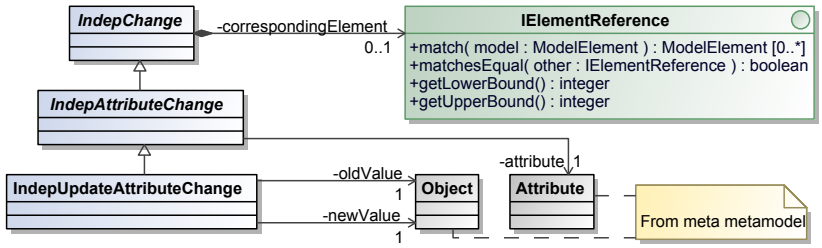


Figure 5.10: The metamodel for changes describing updated attributes.

UML classes [OMG10]. Only updated attributes are explained here (cf. metamodel in Fig. 5.10) because multi-valued attributes are rarely used[4]. However, we decided to include them in order to be symmetric and consistent with reference change types. Again, the following information is required to describe the update of an attribute: *what* attribute was changed and *how*.

What: The symbolic reference of *correspondingElement* defines the model element that owns the updated attribute. *Attribute* specifies which attribute is updated.

How: The value before the change is stored in *oldValue*, the value after the change is stored in *newValue*.

Change #8 in Tab. 5.1 is an example for an updated attribute, the *name* of a UML attribute to be precise. An *IndepUpdateAttributeChange* describes such a change. The corresponding element defines the model element that owns the attribute, the UML attribute *Title* in this case. The updated meta attribute is *name* and the actual change is described by *oldValue = 'Title'* and *newValue = 'title'*.

The same distinction between added/deleted and updated attributes applies as explained for reference change types.

5.3.3 Dependencies

A set of model changes describes what exactly was changed between two versions of a model. When only a subset of model changes shall be applied to other

[4] Just like references, attributes may also have a cardinality and, hence, can be multi-valued.

models, some changes may require the prior application of other changes. The example changes in Tab. 5.1 illustrate the problem: change #5 (adding *IdElement* as a superclass to *Book*) can obviously not be applied before change #1 (adding class *IdElement*) has been applied, because the former change adds a reference to the class that is first created by the latter change. In general, the problem can be stated as follows: *a change C1 depends on another change C2 if C1 is only applicable to a model after C2 has been applied.*

The following dependency relation between changes defines which changes must be applied before others can be applied. This is important in case of partial application of model changes, that is, when only a subset of model changes shall be applied to a model. Whenever a depending change is not contained in that subset, the selected changes cannot be applied successfully. The dependency relation is defined as follows.

- A change *C1* depends on a change *C2*, if *C2* describes the addition of model elements and *C1* contains a reference to any of these added model elements.
- A change *C1* depends on a change *C2*, if *C1* describes the removal of model elements and *C2* contains references to any of these removed model elements.

These two rules are sufficent because of the following reasons. Attribute changes do not correlate with any other change, this is why they are not relevant for dependencies. Adding or updating a reference, adding an element with cross references, or the new parent of a moved element may refer to a newly added element; hence, these changes may depend on the change describing the newly added element. Removing or updating a reference, removing an element with cross references, or the old parent of a moved element may refer to a removed element; hence, the change describing the removed element may depend on these changes. This list covers all supported change types.

5.3.4 Conclusion and Discussion

This section first documented four important design decisions concerning the scope, applicability, internal representation, and the supported set of change types of model-independent differences. These decisions led to the design of model-independent differences that was explained for changed elements, their attributes and references. The entire metamodel of model-independent differences is shown in Appendix A.

Extendibility of these concepts is an important requirement. The first step for adding a new change type is its integration into the metamodel, i.e. to add a new subclass of *IndepChange*, with the desired properties. Afterwards, and that is the major part of the extension, the creation and application of the new change type must be defined. See Sect. 5.6 and 5.8 for details.

The ability to express complex changes is another important requirement. Complex changes might be refactorings like the one presented in the beginning of this section; they basically consist of a set of atomic model changes. Change groups can be used to arrange changes and to structure them hierarchically. Grouping may be performed either at creation-time of the model changes or at any time afterwards. See Sect. 5.7 for details.

5.4 Symbolic References

The metamodel for model-independent differences presented so far covers eleven change types as discussed in the previous section. The specification of *which* model elements are changed, however, was only vaguely mentioned as symbolic references. There are many different *matching strategies* in the literature, each having its own strengths and shortcomings. Instead of restricting the metamodel to one strategy only, we define an interface which allows the use of different matching strategies. This section explains the concepts of symbolic references along with three example realizations: id-, condition-based, and static referencing of model elements.

The most important characteristic of symbolic references is that they define the matching strategy during the application of model changes. That is, during change application (step 4 in Sect. 5.2), each symbolic reference *matches* elements in the target model. The results of all matchings yields the initial mapping Ψ that maps each symbolic reference to the set of model elements it matches. We first discuss the meaning of symbolic references before defining the interface and realizations.

A widely used and simple solution are unique identifiers where a (single) model element with a given identifier is matched. Typically, identifiers are unique in the scope of the model that contains the model element, in contrast to *globally* unique identifiers, which are unique even across other models. However, not all metamodels provide unique identifiers. Other solutions are similarity metrics or model queries. Overall, we call the parts that define which elements are matched in a model *symbolic references*. The term is inspired by Venvers:

> "A symbolic reference is a character string that gives the name and possibly other information about the referenced item – enough information to uniquely identify [it]." [Ven99, p. 56]

However, we do not limit symbolic references to character strings but we retain the characteristics that symbolic references are self-contained (as a character string is) and it must contain some information to (not necessarily uniquely) identify the referenced items. Consequently, a symbolic reference must in particular not depend on any other artifact like other models.

Let us assume a model change describing the addition of a generalization shall be applied to *all classes in a package "data"*. Moreover, there shall be at least one and at most three such classes: *[1..3]*. The subsequent sections discuss how such a model reference can be matched to the unchanged version of model M_A in Fig. 5.1 on page 36.

5.4.1 The Interface for Symbolic References

A symbolic reference must provide a *matching algorithm* that matches all elements in a given model which are potential candidates. Symbolic references may be used at many places, so they do not know about any model change they belong to. A matching algorithm may in particular return a *set of model elements*. Hence, the signature of the algorithm is:

match(model : ModelElement) : ModelElement [0..∗] (cf. metamodel in Fig. 5.11)

The parameter *model* is the root element of the given model or, to reduce complexity and to improve performance, of a sub-model.

For each symbolic reference, bounds must be set which specify whether or not the number of matched model elements is valid (cf. the interface *IElementReference* in Fig. 5.11). Matching the symbolic reference *all classes in a package "data"* to the unchanged version of model M_A in Fig. 5.1 on page 36, returns four classes, *Article*, *Book*, *Cover*, and *Author*. According to the specified bounds *[1..3]*, the size of this set is not valid. The refinement of the mapping (step 5 in Sect. 5.2) must resolve that issue by removing at least one of the classes from the set.

5.4.2 Matching Strategies – Package *symrefs*

This section presents several realizations of symbolic references including their matching strategies. An overview of existing matching strategies in the literature and related work is given in Tab. 5.8. Each of the existing approaches,

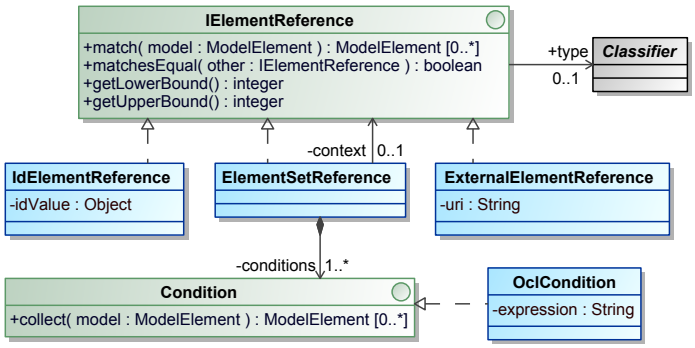


Figure 5.11: The metamodel for symbolic references and three example implementations (package *symrefs*).

however, allows only one specific matching strategy at a time; since model-independent differences specify the matching strategy per symbolic reference, several different strategies are possible even in the same set of model changes.

The table lists often used matching strategies from literature along with their pros and cons. Id-based matching is typically fast and the only one that matches at most one element. All others are capable of matching multiple model elements although they often return only a single element, for instance, the most similar one in case of similarity-based matching. In order to make model-independent differences applicable to other models as motivated in the example in the beginning of this Sect. 5, we would like to *configure* the matching strategy or, in other words, manipulate the matching criteria. All matching strategies but the id-based one are to a greater or lesser extend configurable.

The package *symrefs* contains three implementations of symbolic references as shown in Fig. 5.11. Two of them (*IdElementReference* and *ExternalElementReference*) comprise id-based matching strategies, whereas *ElementSetReference* comprises a condition-based matching strategy. Each of them is presented next with a brief description, an outline of the matching algorithm, and a discussion of their advantages and disadvantages.

Id-based Symbolic References. This strategy of referencing model elements requires a distinguished property as identifier on each model element that is unique within its model. This is a common and efficient way of referencing model elements. The proposed realization *IdElementReference* stores the value of such a unique identifier in *idValue*. The upper bound is always 1.

Type	Uses	Description
Id-based *	[OWK03] [AP03] [BP08]	Each model element has an identifier that is either unique within its model or globally unique. A symbolic reference matches the model element with a particular identifier.
Adv.:		+ Exact and very fast
Disadv.:		− Unique identifiers are not available in all models − Independently created models have different identifiers − It is not possible to match more than one element − Configurable only by adjusting the id value
Similarity-based	[TBWK07] [BP08] [LGJ07]	A reference model (e.g. copy of original model) is used to match the model element of interest. Usually attribute values and/or graph structure are compared.
Adv.:		+ No unique identifiers required + Generically applicable + Configurable by adjusting the similarity threshold + Possibility to match a set of elements
Disadv.:		− Weak performance ($n * m$ comparisons in the worst case; n = #elements in reference model, m = #elements in target model) − False positives might occur in case the model differs from the original model − Configuration: adjusting the reference model is cumbersome
Query- or signature-based	[FBFG07]	A user-defined function is used to query the model which returns a set of elements with particular properties (defined in a signature). Performance depends on the queries.
Adv.:		+ Highly configurable + Returns a set of multiple elements
Disadv.:		− A signature specification is required for each type of model elements − Ambiguity (matching more than one element) may not always be desired
Condition-based *	[BLS+09]	This specialization of signature-based referencing uses conditions as signatures.
Adv.:		+ Configurable by adjusting the conditions + Standardized query languages are available (e.g. SQL [GWO09], OCL [OMG06b]) + Signature can easily be generated
Disadv.:		− Performance depends on query execution engine − Ambiguity (matching more than one element) may not always be desired

Table 5.8: Different matching strategies used in related work; matching strategies marked with a (*) are also realized in this project.

Examples:

idValue="_wK5TIkMyEd−jXvz6ingaZQ"	(value of id attribute in a UML model)
idValue="324"	(value of id attribute in an EMF model)

Matching Algorithm: If the given model contains a model element with the identifier stored in *idValue*, that model element is returned. Otherwise an empty set is returned.
Advantages: The algorithm is fast (it typically runs in *O(1)* with a one-time calculation of *O(n)*, *n* being the total number of elements in the model). By definition of unique identifiers, its return value is unambiguous, that is, at most one element.
Disadvantages: Id-based references are not suited for applying changes to other models that are created independently of the original model, because the other model elements usually have different identifiers assigned.

Static Symbolic References. This specialization of id-based symbolic references uses *absolute*[5] URIs (*Uniform Resource Identifier* [BLFM05]) for referencing model elements. Compared to the preceding id-based symbolic references, static symbolic references use URIs as absolute paths whereas the former may use relative paths to the model root. Hence, URIs can be used to refer to static model elements such as data types in UML profiles. A URI consists of several parts, those before the fragment (separated with '#') typically locate the model. The fragment is typically the unique identifier or path to the element of interest within that model. Since such referenced elements are typically located in other than the compared models, our realization is called *ExternalElementReference*.
Examples:

uri="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.uml#String"	(UML profile)
uri="platform://resources/example/library.ecore#//Book/id"	(EMF model)

Matching Algorithm: The algorithm is the same as the previous one with a preceding loading of the model specified in the URI. The parameter *model* is not used here.
Advantages: In addition to the advantages of the previous strategy, static model elements outside the target model can be referenced.
Disadvantages: Only one single fixed model element can be referenced.

Condition-based Symbolic References. This is a mixture between query- and similarity-based referencing strategies (cf. Tab. 5.8). The proposed realization (cf. *ElementSetReference* in Fig. 5.11) uses *Condition*s that must hold for referenced model elements. These conditions contain boolean expressions and can, for instance, be expressed in OCL (*Object Constraint Language* [OMG06b]).

[5] The specification of URIs allows absolute and relative paths, starting from a context element; however, static symbolic references only use absolute paths.

Conditions may check attribute values, graph structure, or any other properties. In contrast to signature-based model referencing, we create conditions automatically from the original model elements (more details are given later in Sect. 5.6). A *context* may be used to restrict the matching to a sub-model in order to reduce complexity and to increase performance.
Examples:

```
expression="self.name='data'"                                  (UML package)
expression="self.name='Book' and self.isAbstract=false and..."  (UML class)
```

Matching Algorithm: For each condition, *collect* is called and returns all elements for which that condition holds[6] (cf. metamodel in Fig. 5.11). The intersection of the results from all conditions yields the overall matching result. This may in particular be a set of multiple model elements. If the *context* is set, then the conditions are not evaluated on the entire model but on the matching result of the context.
Advantages: The matching strategy is very flexible, any kind of conditions may be used here. Users may also manually refine automatically created conditions.
Disadvantages: Checking the conditions on each model element during the matching process might be a time-consuming operation.

5.4.3 Conclusion and Discussion

This section motivated the term *symbolic references* and presented its interface. Several matching strategies have been presented that are used in the literature and that could be integrated into model-independent differences; id-, condition-based, and static symbolic references, have already been integrated.

Although the default cardinality for the matching result of symbolic references is *1..1*, all but id-based realizations can potentially match multiple model elements. That means, model changes could be applied to multiple model elements instead of just one. However, except for the work by Brosch et al. [BLS+09], none of the related work exploits this feature.

The remaining parts of the thesis focus on condition-based symbolic references because conditions can easily be manipulated. This fact will be used later in Sect. 5.7 to generalize atomic changes to make them applicable to other models. This is not possible with id-based and only hardly possible with similarity-based model referencing strategies. Id-based model referencing, on the other hand, is the best choice for the traditional way of creating and applying a patch, which is also supported by model-independent differences.

[6] For OCL conditions, an OCL engine evaluates the expression on the model elements; no extra traversal algorithm is required.

5.5 Model Descriptors

Symbolic references can be used to refer to arbitrary model elements without the need of any other model. That could, for instance, be the place *where* new model elements are added. But how to express *what* to add? Since model-independent differences shall be independent of any model (including the original model from which they have been created), all information about the changes must be included in the differences, including the model elements that have been added.

A description of an added element must include all sub-elements and all their properties including attribute values and references to other model elements. A copy or naive serialization of the added parts is not sufficient, because information about cross references to other model elements will be lost. Holding a copy of the entire compared models is impractical especially in case of big models. Others use unique identifiers or URIs to describe cross references, but such descriptions are not applicable to models with different contents.

This section continues with a classification of cross references on an extended example in order to highlight the problem of describing added parts of a model. Then the interface for model descriptors is presented that is capable of describing added model elements sufficiently such that they are again applicable to other models. The section concludes with a discussion about the presented concepts.

5.5.1 Cross References in Model Descriptors

The crucial part of describing elements that have been added to a model is to store cross references between these model elements and the rest of the model. Fig. 5.12 is a variation of the change that adds a class *IdElement* to a model; it does not only contain an attribute *id : String* but in addition an inheritance relation to an existing class *IdGenerator*. There are two changes, *add generalization* from *Book* to *IdElement*, and the already mentioned addition of *IdElement*. We must consider three types of cross references when adding elements to a model, all of them are covered in this example. See Tab. 5.9 for an overview.

In order to be applicable to other models, the description of cross references must be detached from the originally compared model. Therefore, symbolic references can be used for describing the different kinds of cross references:
cross references between changes: internal element references,
cross references to model elements: id- or condition-based symbolic references,
cross references to external elements: static symbolic references.

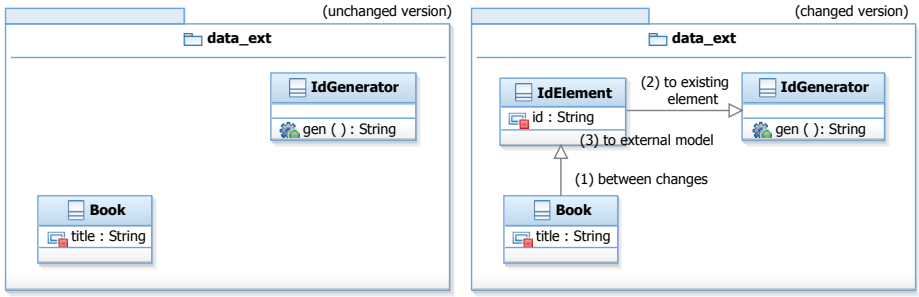


Figure 5.12: Several kinds of cross references might occur in model changes: cross references within changes, to model elements, or to external models.

Cross reference	Description
(1) between changes	A change, e.g. an added reference, refers to a model element that is added to the model.
(2) to existing model elements	A change, e.g. an added element, refers to an existing model element.
(3) to external model elements	A change, e.g. an added element, refers to a model element that is located in another than the compared model.

Table 5.9: List of different types of cross references that may occur when describing added model elements.

The latter two types of cross references have already been discussed in Sect. 5.4. *Internal element references* are references to model elements that do not exist in a model unless the according change has been applied that creates that model element. These references are explained later in Sect. 5.5.4 after the interface for model descriptors has been presented.

5.5.2 The Interface for Model Descriptors

Our proposed concepts for describing added model elements sufficiently is called *model descriptors*. The interface *IModelDescriptor* in Fig. 5.13 requires, on the one hand, that realizations store all cross references properly. On the other hand, it requires that realizations are able to apply described model elements to another model, as explained next.

Added elements might refer to each other as sketched in Fig. 5.14. The figure illustrates two changes, an added element A including a reference to B and an added element B including a reference to A. Assume both changes shall be ap-

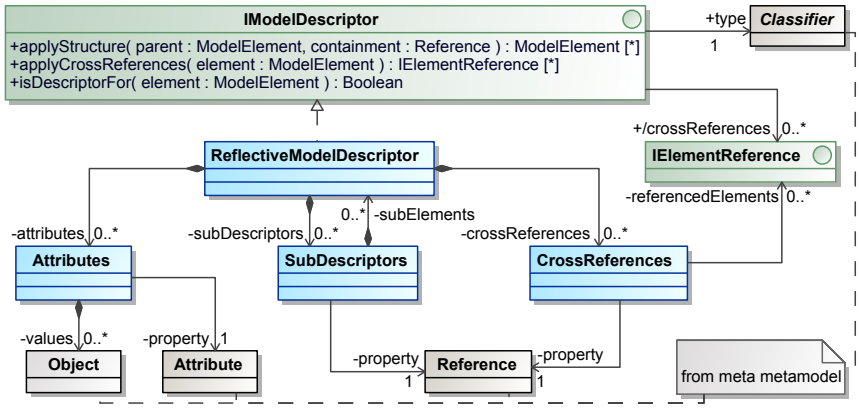


Figure 5.13: The metamodel for model descriptors and a realization for reflective models (package *descriptors*).

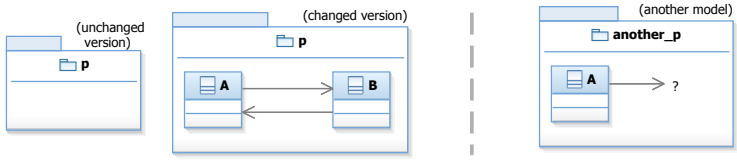


Figure 5.14: Two changes that depend on each other; the application of add element changes requires two steps: first, all elements are added; second, all cross references are restored.

plied to another model, then none of them is applicable individually because the reference to the other element cannot be restored. A solution is to perform the application of added elements in two steps: first, the described model elements including their attributes are created in the target model; second, all cross references are restored. These two steps are reflected in the two operations *applyStructure* and *applyCrossReferences* of the interface *IModelDescriptor*.

5.5.3 A Generic Model Descriptor

The proposed realization of a model descriptor *ReflectiveModelDescriptor* uses reflection to access and restore properties on model elements. Three types of information must be stored: attributes values, cross references, and sub-elements. Each of them is described as depicted in Fig. 5.13:

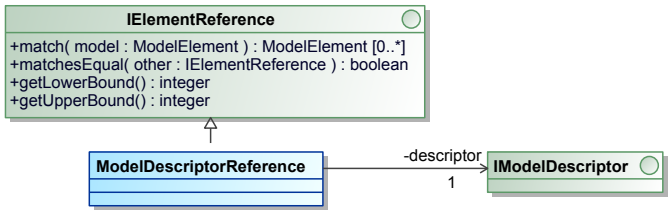


Figure 5.15: The metamodel for symbolic references that refer to model elements which are described by a model descriptor.

- *attributes* contains a list of attribute values for a property (*Attribute*);
- *subDescriptors* contains a list of sub-element descriptors for a property (*Reference*);
- *crossReferences* contains a list of symbolic references (the actual referenced model elements) for a property (*Reference*). *IModelDescriptor.crossReferences* is an accumulated set of all cross references including sub elements.

This is all information required to apply a model element to a model: all sub-elements and all attributes and references can be restored, assuming that the symbolic references are resolved properly. The latter is, however, part of step 5, the resolution and refinement of symbolic references, as outlined in Sect. 5.2.

5.5.4 Internal Model References

In addition to the symbolic references presented in Sect. 5.4, we need another type of model references, namely references to model elements that have not yet been added to a model. Changes #5–7, for example, describe adding a generalization from existing model elements to the not yet existing element *IdElement*. Id- or condition-based references will not work, because they cannot match an element that does not yet exist in a model. Going through all cases of referencing model elements within model changes, there are three cases in which such references may occur:

- Added/updated references that refer to added model elements
- Removed/updated references that refer to deleted model elements
- Cross references within the same or across model descriptors

The targets of these model references are always elements described by model descriptors. Our consequent realization of these particular symbolic references is called *ModelDescriptorReference* and is shown in Fig. 5.15.

The matching strategy of these symbolic references is obvious: they match exactly those model elements that have been created by the referenced model descriptor. This implies that these symbolic references do not return concrete model elements during the matching phase; but they return the added elements after the first phase during the application step (applying structure of added elements, cf. Sect. 5.5.2).

5.5.5 Conclusion and Discussion

This section presented the requirements, concepts, and a realization of how model elements including sub-elements can be described in model changes. The crucial part of these model descriptors is to store cross references properly such that they can be restored whenever the described element is applied to a model. The solution is the use of appropriate symbolic references.

There is not much information about related work for describing model elements including cross references; Cicchetti et al. [CRP07] and Hermannsdörfer et al. [HK10] bypass that issue by using separate add reference changes for cross references, Brosch et al. [BLS+09] use copies of the compared model fragments, and all others refer to the originally compared models.

The concept of model descriptors are again designed in a modular way in case other realizations will be developed in the future. Imaginable are specific model descriptors for models that do not support reflection or model descriptors that work without symbolic references.

5.6 Creating Model-independent Differences

The previous sections introduced the metamodel of model-independent differences and explained how they are used. Their creation was only briefly mentioned in step 2 of Sect. 5.2, which said that they are the result of a transformation from another model-dependent differencing format. After giving an example of model-dependent differences, this section discusses the requirements for the creation task and then introduces a transformation using QVT Operational Mappings from differences of EMF Compare into the format of model-independent differences.

The use of EMF Compare is one example for model-dependent differences. We could use others, for instance, the operation recorder from Hermannsdörfer et

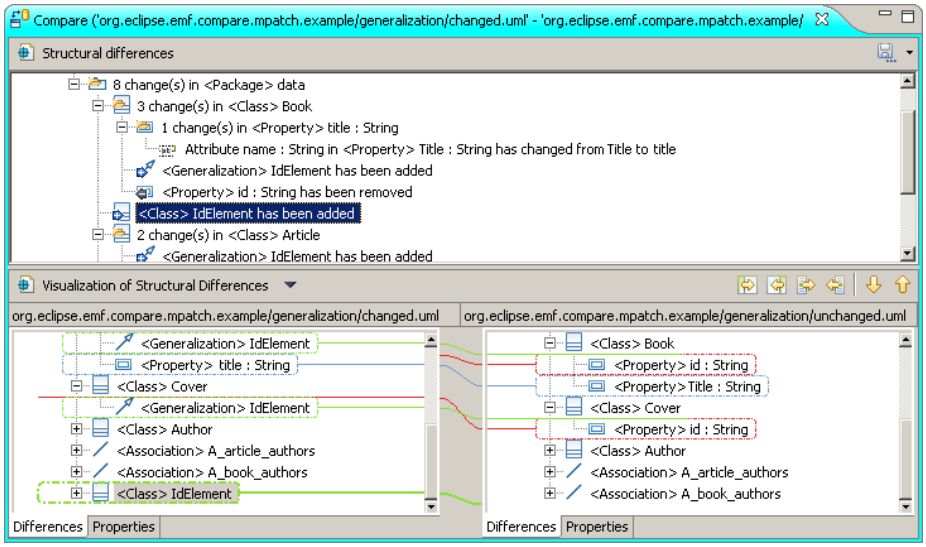


Figure 5.16: EMF Compare calculates and visualizes all changes between the two versions of model M_A shown in Fig. 5.1: a newly added class, three deleted attributes, three added generalizations, one updated attribute.

al. [HK10] which hooks into model editors and records all model changes. Such recorded model changes are much more precise than those created with EMF Compare, but the model must always be edited with specific editors. Those other sources are imaginable as well and may be added in the future.

5.6.1 Example of Model-dependent Differences

Eight changes have been made to the model in Fig. 5.1 on page 36 as part of the refactoring *extract superclass*. Fig 5.16 depicts them in the GUI of EMF Compare. The upper part summarizes all changes, the lower part visualizes the changes in a tree-based view of the model. All eight changes are shown: a new class *IdElement* has been added to the package *data*, generalizations have been added and attributes have been removed for three classes, and the attribute *Title* was renamed.

These are *model-dependent* differences because the unchanged and the changed model versions are required to represent and visualize the changes. Fig. 5.17 shows the addition of the class *IdElement* in abstract syntax as a UML object

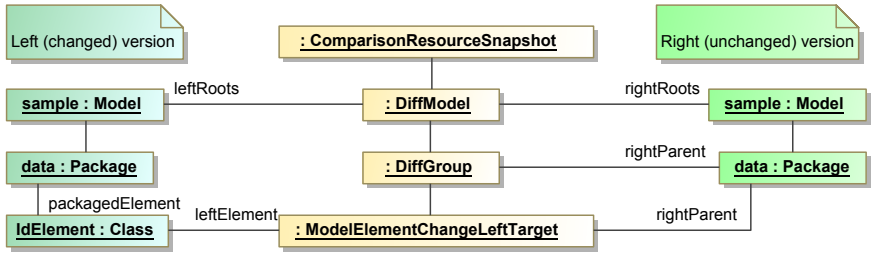


Figure 5.17: Parts of the model changes in Fig. 5.16 are shown here in abstract syntax as a UML object diagram, its metamodel is given in Fig. 5.18; this description of model changes refers directly to the compared model versions.

diagram. The *ComparisonResourceSnapshot* specifies that two resources have been compared and it contains a *DiffModel*. The latter contains the actual changes, hierarchically structured corresponding to the containment hierarchy in the unchanged version of the model that is compared. *ModelElementChangeLeftTarget* specifies that the *left* model version, which is the changed version of the model, had an *element change* – which means that an element has been added to the model. The added element is referenced via the association *leftElement*, its parent via *rightParent*.

In total, EMF Compare supports ten types of changes (cf. Sect. 3.2), all are represented similarly: direct references to model elements specify *which* model elements have been changed and the meta type (*ModelElementChangeLeftTarget* in this case) specifies *how* it was changed. The meta type of the root container (here *ComparisonResourceSnapshot*) specifies whether two single resources or a set of resources, e.g. multiple files, have been compared.

5.6.2 Transformation Requirements

This section explains the requirements for the transformation to create model-independent differences so that they can be used as illustrated in the scenarios in Sect. 5.2. Both model-dependent and model-independent differences, can be treated as models. Therefore, we define a model-to-model transformation for the creation of model-independent differences. The requirements as well as the source and target metamodels for this transformation are stated and the chosen transformation language is discussed below.

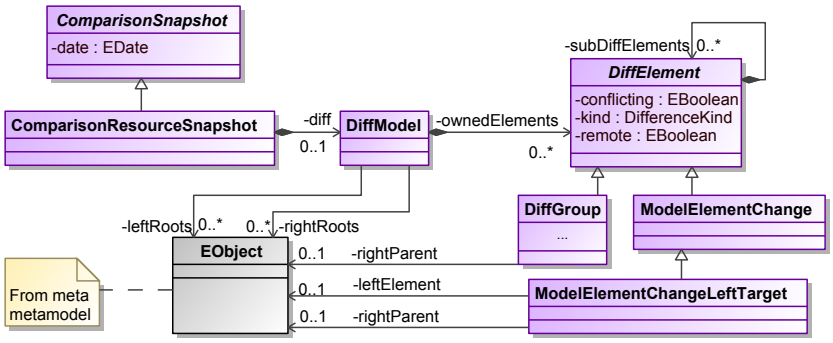


Figure 5.18: An excerpt of the EMF Compare metamodel that is the source metamodel for the transformation θ_1 to create model-independent differences.

The transformation θ_1 to create model-independent differences requires a valid set of model-dependent differences Δ as input. *Valid* means that all referenced elements must exist, including the unchanged and the changed versions of the compared model. Moreover, there must not be a conflicting model element (conflicts might occur in three-way model comparisons, which is neglected here). Then θ_1 transforms the model-dependent differences Δ including the unchanged and changed versions of the model M_A and M'_A into self-contained model-independent differences: $\Delta_a = \theta_1(\Delta, M_A, M'_A)$. The model-independent differences Δ_a must contain all information about the changes made to M_A such that M'_A can be restored from M_A: $M'_A = \pi(\Delta_a, \psi_1(\Delta_a, M_A), M_A)$ and vice versa; as a reminder, ψ_1 is the matching, π the application of model differences as defined in Sect. 5.2.

There are also two important non-functional requirements to the transformation: *automation* and *extendibility*. The transformation must be automatic, i.e. no user interaction must occur, because it should be possible to provide the creation of model-independent differences via a public API[7]. Furthermore, the transformation must be easily extendable, for instance, for adding additional change types or new matching strategies.

The source metamodel for the transformation is the EMF Compare metamodel. Fig. 5.18 shows an excerpt of it, the full metamodel is shown in Appendix A. The object diagram in Fig. 5.17 is an instance of this metamodel. The excerpt covers resource-based model differences (*ComparisonResourceSnapshot*) and the change type for added elements (*ModelElementChangeLeftTarget*). The

[7] An *Application Programming Interface* (API) can be used by third party application to access provided functions.

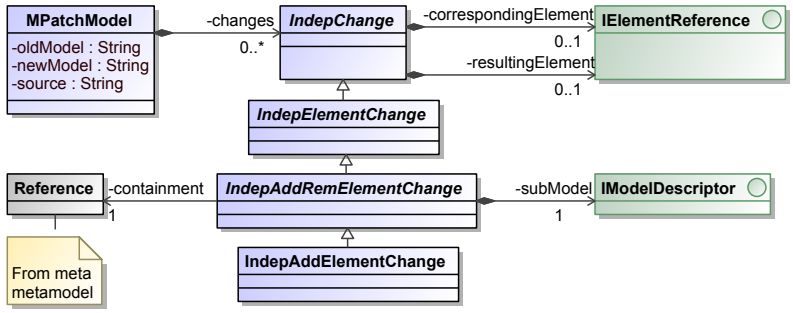


Figure 5.19: An excerpt of the metamodel for model-independent differences, the target metamodel for the transformation θ_1.

subclasses of *DiffElement* represent the individual change types. Both attributes *conflicting* and *remote*, are irrelevant because they state properties for three-way comparisons and comparisons of models in repositories, respectively. References to *EObject*, on the other hand, are important because they refer to the actual changed elements in the compared models. No information of the change itself is described with this metamodel, it is derived information from the referenced model elements. An added element, for instance, is expressed indirectly via the reference *leftElement* which points to the added element in the changed version of the compared model (as shown in Fig. 5.17).

The target metamodel for the transformation is the one defined in Sect. 5.3. Fig. 5.19 shows an excerpt of it with all relevant elements to understand the transformation of added elements. *MPatchModel* is the container for all changes. *IndepChange* is an abstract superclass for all change types, similar to *DiffElement* in the source metamodel. Its concrete subtypes encode the change type, just as the source metamodel. Direct references to the compared models are, in contrast to the previous metamodel, represented with the two interfaces *IElementReference* and *IModelDescriptor*.

Before explaining the transformation specification, we informally sketch the transformation for the example in Fig. 5.17. Figure 5.20 shows the source instance at the top and a target instance at the bottom. The arrows sketch the actual transformation: *DiffModel* is transformed into an *MPatchModel*, *ModelElementChangeLeftTarget* into *IndepAddElementChange*. *PackagedElement*, the property that contains the added element, is transformed into a reference to *packagedElement : Reference*, which is defined in the UML metamodel. The creation of model descriptors (denoted with *MD*) and symbolic references (denoted with *SR*) are modular and exchangeable operations, because there might be dif-

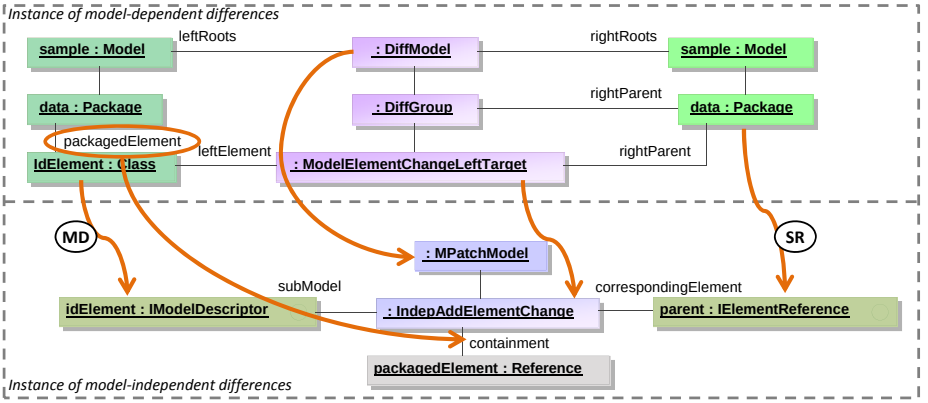


Figure 5.20: Informal sketch of the transformation of model-dependent differences from Fig. 5.17; the arrows denote the actual transformation; *MD* states the creation of a model descriptor, *SR* the creation of a symbolic reference.

ferent realizations for these two concepts (cf. Sect. 5.4 and 5.5).

Based on these requirements, the chosen transformation language is QVT Operational Mappings [OMG07a]. Most importantly, the language allows the import of libraries for exchangeable modules for the creation of model descriptors and symbolic references. Moreover, the language constructs allow an easy and elegant way of defining short and concise mappings for each change type. Details are stated in the following section.

5.6.3 Transformation Specification: θ_1

This section outlines the transformation θ_1 of model-dependent differences calculated by EMF Compare into model-independent differences. The entire transformation specification in QVT is given in Appendix A.

The transformation specification consists of a set of transformation mappings, starting with a root mapping and one mapping for each change type. Tab. 5.10 gives an overview of all mappings. The mapping *toIndepChange* is overloaded and exists for each change type – the source model element type defines which concrete mapping is used. The mappings themselves are listed in the appendix. Figure 5.20 illustrates the application of two mappings, namely *toMPatchModel* to create the root element of model-independent differences, and *toIndepChange* for the type *ModelElementChangeLeftTarget*.

Mapping	Source model element	Target model element
toMPatchModel	*DiffModel*	*MPatchModel*
toIndepChange	*DiffElement*	– (redirected to *toUnknownChange*)
toIndepChange	*ConflictingDiffElement*	– (transformation aborts)
toIndepChange	*MoveModelElement*	*IndepMoveElementChange*
toIndepChange	*ModelElementChangeRightTarget*	*IndepRemoveElementChange*
toIndepChange	*ModelElementChangeLeftTarget*	*IndepAddElementChange*
toIndepChange	*UpdateAttribute*	*IndepUpdateAttributeChange*
toIndepChange	*AttributeChangeLeftTarget*	*IndepAddAttributeChange*
toIndepChange	*AttributeChangeRightTarget*	*IndepRemoveAttributeChange*
toIndepChange	*UpdateUniqueReferenceValue*	*IndepUpdateReferenceChange*
toIndepChange	*ReferenceChangeLeftTarget*	*IndepAddReferenceChange*
toIndepChange	*ReferenceChangeRightTarget*	*IndepRemoveReferenceChange*
toUnknownChange	< *any* >	*UnknownChange*

Table 5.10: Overview of all mappings of the transformation specification.

Except for the creation of model descriptors and symbolic references, the mappings are straightforward; most elements are transferred one-to-one. Two operations *toSymbolicReference()* and *toModelDescriptor()* are used by the mappings to create symbolic references and model descriptors, respectively (denoted by the arrows *MD* and *SR* in Fig. 5.20). They are not part of the actual transformation but delegated to an imported library. The latter is explained next.

5.6.4 Transformation Configuration and Extensions

The transformation is configurable in terms of the selection of symbolic references and model descriptors. Both cannot be created with QVT Operational Mappings because the essential reflective operations for accessing element properties (*get(Property) : Object*, defined for MOF-based models [OMG06a]) are not available. A blackbox library provides these two operations:

EObject.toSymbolicReference() : IElementReference

EObject.toModelDescriptor() : IModelDescriptor

The library does not implement the logic of these operations itself but delegates them to other modules that must be specified for each transformation. These modules are extensions to the transformation as explained below.

Creating Symbolic References. Symbolic references define the matching strategy of the created model-independent differences when applying them to a model. A *symbolic reference creator* is a module that realizes the operation *toSymbolicReference* to create a symbolic reference for a given model element. As mentioned in Sect. 5.4, three concrete realizations are already supported.

The one creating static symbolic references is automatically used whenever the given model element is not contained in the model on which the transformation was started. Either of the other symbolic reference creators must be used for all other model elements. Instead of showing the code for each module, the following outlines the algorithms that create the respective symbolic references.

- *Static symbolic references:* an instance of *ExternalElementReference* is created and *uri* is set to the URI of the given model element.
- *Id-based symbolic references:* an instance of *IdElementReference* is created and *idValue* is set to the value of the attribute that is the unique identifier of that element. If the element does not have a unique identifier, the fragment of the element's URI is used instead.
- *Condition-based symbolic references:* an instance of *ElementSetReference* is created and conditions are added that contain sufficient information for matching the given model element. By default, the conditions consider only attribute values, assuming that the set of all attribute values identifies the model element sufficiently[8]. Moreover, another *ElementSetReference* is created as *context* for the container of the given model element.

Example conditions of condition-based symbolic references for the class *Book* in Fig. 5.1 are:

```
self.name='Book'and self.isAbstract=false and self.isActive=false and self.isLeaf=false
and self. visibility =VisibilityKind::public
```

The default conditions check all attributes values. In addition, the *type* must match, which is a UML *Class*. The context is another symbolic reference to the container, the package *data*, having the following condition:

```
self.name='data'and self. visibility =VisibilityKind::public
```

That condition involves only two attributes because UML packages do not own other attributes.

Since the symbolic reference creator is a module in this framework, it can be exchanged, for instance, by replacing it with another condition-based symbolic reference creator that considers the graph structure or other model-specific attributes.

Creating Model Descriptors. Model descriptors defines how added and removed model elements are stored within model changes. Again, a *model descriptor creator* is a module that realizes the operation *toModelDescriptor* to

[8] In contrast to id-based symbolic references, this condition might match more than a single model element; depending on the usage scenario, all these other elements may be false positives that must be eliminated during the refinement (step 5, Sect. 5.2).

create such a model descriptor for a given model element. The only realization at the moment is the *ReflectiveModelDescriptor*, whose creation is briefly sketched with the help of the example.

A model descriptor for the model element *IdElement* in Fig. 5.1 on page 36 is created as follows. First, an instance of *ReflectiveModelDescriptor* is created (cf. the metamodel in Fig. 5.13 on page 63). Then all attributes (*name*, *isAbstract*, *isActive*, *isLeaf*, and *visibility*) and their values are stored in *attributes*. For each referenced model element, an appropriate symbolic reference is created using symbolic reference creators and stored in *crossReferences* (the class *IdElement* does not have cross references but its sub element *id : String* has a cross reference to the type *String*, located in an external UML profile). Finally, the model descriptor creator is recursively called on all sub elements (UML attribute *id*) and their result is stored in *subDescriptors*.

This way, entire models can be stored in model descriptors and restored from them without losing any information.

5.6.5 Additional Transformations

So far, a transformation produces an unstructured set of model changes. One important part, the dependency graph between model changes (cf. Sect. 5.3), is still missing. Further properties like structured model changes might be desired, depending on the scenario. To this end, additional in-place transformations on the initially created model changes can be performed to incorporate such properties. The dependency graph is the only mandatory transformation and explained next, all other transformations are optional and explained later in Sect. 5.7.

The dependency graph is created according to the dependency rules stated in Sect. 5.3. These rules describe a directed graph on model changes, may contain cycles, and is not necessarily connected. The graph is stored in the bidirectional relation *dependsOn/dependant* of *IndepChange* (see metamodel in Fig. 5.6). The algorithm for creating the dependency graph analyzes all cross references to model descriptors in order to identify dependencies that are required in the dependency rules. Having all these dependencies collected, the realization of the algorithm is straightforward.

5.6.6 Conclusion and Discussion

The creation of model-independent differences has been presented as a transformation from model-dependent differences. The relevant parts of the transformation specification have been explained to understand the transformation, the full specification is given in Appendix A.

The creation of model-independent differences depends on the selection of symbolic references, because different symbolic references comprise different matching strategies during their application. For this reason, the selection of symbolic references must be configured. Furthermore, the transformation is extendable in case new symbolic references or model descriptors are defined, and appropriate creator modules can be added to the transformation.

The dependency graph (required for the partial application of model changes) can only be calculated after all changes have been created. Therefore, the calculation of dependencies takes place in an in-place transformation that is performed after the main transformation. We use two static rules for calculating the dependencies, dynamic approaches [GKLE10] might be interesting alternatives. Moreover, the initial set of model changes is unstructured, in contrast to most other differencing technologies [BP08, CRP07, TBWK07, XS05]. Groupings or other strategies of structuring the differences can also be performed via optional transformations, which is discussed in the next section. We have chosen this separation to make the creation of model-independent differences as modular and flexible as possible.

5.7 Generalizing and Structuring Model Changes

The creation of model-independent differences results in an unordered set of model changes; the only property that is set so far is the dependency graph. This section first motivates the generalization and structuring of model-independent differences before explaining heuristic algorithms for both aspects.

The motivation for raising the *abstraction* level of calculated atomic model changes was already given in the example on the first pages of this Sect. 5. If model differences calculated from example models are applicable to other models as well, then users can save a lot of manual modeling effort by applying the generalized model differences to the other models. *Structuring*, on the other hand, does not at all alter the applicability of model differences, but it improves readability (and, hence, usability) of model-independent differences.

An overview of all generalization and structuring transformations are listed in Tab. 5.11 and each of them is explained below.

Transformation	Description
θ_S: Scope Expansion	Makes model changes applicable to model elements that are *similar* to the original one.
θ_U: Unbounding References	Makes model changes applicable *multiple times*.
θ_M: Merging Changes	*Merges* similar model changes into a single generic change that is applicable to multiple model elements.
θ_G: Grouping	Introduces *groups* for unstructured model changes.
θ_R: Reversal	*Reverses* the direction of model-independent difference application.

Table 5.11: An overview of our generalization (θ_S, θ_U, θ_M) and structuring (θ_G, θ_R) transformations.

5.7.1 Scope Expansion: θ_S

The purpose of *Scope Expansion* is to add fuzziness to the matching process. That is, the scope of a change, which denotes the set of model elements that can potentially be matched, will be extended. Consequently, also slightly different model elements could be matched. The valid number of matched model elements is not changed, scope expansion only intends that the matched element is similar and not necessarily equal to the original one. For similarity-based model references, for instance, scope expansion could mean to decrease the threshold when two elements are considered being the same; e.g. 50% instead of 90% of the elements' properties must be equal.

For condition-based references, which check the elements' attributes (cf. Sect. 5.4), we propose a modification of the OCL conditions: the equality checks of string attributes is replaced with a similarity check. The heuristic algorithm is based on two rules:
Rule 1: containment is important.
Rule 2: case sensitivity is not important but should neither be neglected.
Then the similarity $s \in [0, 1]$ between two strings has the following meaning:

If $1 = s$	The two strings are equal.
If $1 > s > 0.5$	One of the strings is a substring of the other: the higher s, the more similar the two strings are.
If $0.5 > s \geq 0$	The similarity is calculated by the Levensthein distance[9].

[9] An algorithm that calculates the similarity of two strings. See Appendix B for details.

In general, the higher the value, the more similar the strings are. The intervals were specified due to our experiences with refactorings and design patterns (see also Sect. 8.1.1). The full algorithm is listed in Appendix B. Example results of the algorithm are shown below:

id	id	→ 1.0	✔	data	DATA	→ 0.9	✔
id	mid	→ 0.85	✔	data	data2	→ 0.88	✔
id	cont_id	→ 0.72	✔	data	customerdata	→ 0.75	✔
id	name	→ 0.0	✗	data	date	→ 0.37	✗
id	middleman	→ 0.69	✗	season	person	→ 0.33	✗
id	blind	→ 0.2	✗	observer	subjectObserver	→ 0.72	✔

Two strings, the calculated similarity, and the comparison result for a threshold of $t = 0.7$ are listed. Using a higher similarity threshold would reduce the number of false positives while a lower threshold would match more potentially relevant model elements. Based on our experiments, we have chosen a threshold of $t = 0.7$ as an appropriate compromise between successfully matched model elements and false positives.

The symbolic references for attributes *id : String* in Fig. 5.1 on page 36, for example, is changed from

```
expression="self.name='id' and self.isDerived=false and..."
```

to

```
expression="self.name.isSimilar('id',t) and self.isDerived=false and..."
```

with $t = 0.7$. So the new symbolic reference matches *id*, *mid*, and *cont_id*, but not *name*, *middleman*, or *blind*. In the end, however, a manual review of the initially matched set of model elements is always recommended.

5.7.2 Unbounding References: θ_U

Symbolic references based on unique identifiers match the single model element that has the given identifier. Condition-based symbolic references, in contrast, are able to match *a set of* model elements instead of just one. In order to make a change applicable to multiple model elements, its upper bound must be removed. Some example scenarios are:

- An attribute is deleted from *several* classes.
- A generalization is added to *several* classes.
- *All* classes in one package are moved to another package.

That is done by changing the bounds of symbolic references from *[1..1]* (default) to *[1..*]* (cf. *upperBound* in the metamodel in Fig. 5.11 on page 57). The transformation θ_U only changes the cardinality of the symbolic references which define *where* changes are applied (*IndepChange.correspondingElement* and *IndepChange.resultingElement*).

However, for some changes it does not make sense to automatically unbound the cardinality; for instance, if a class that is supposed to exist only once in the entire model is added to multiple packages. Then this issue must be resolved in the refinement step 5 (cf. Sect. 5.2).

5.7.3 Merging Changes: θ_M

This transformation merges similar changes into single, generic changes. This reduces the number of changes in order to make the model differences more concise. Moreover, generic changes are applicable to many more model elements, in contrast to unmerged model changes. The algorithm works as follows:
Search for two or more changes that are equal except for the symbolic reference corresponding element *that specifies* where *the changes are applied. Replace these changes with one that combines the differing symbolic references.*
For condition-based symbolic references, the replacement is done by keeping those conditions that all changes have in common and removing all other conditions. Id-based symbolic references cannot be merged.

Regarding the example, the addition of a UML generalization to three elements (changes #5–7 in Tab. 5.1) can be merged into a single change. These are the conditions for the individual changes before the merge (t being the threshold):

name.isSimilar('Article',t) and isAbstract=**false** and ... (type: Class)
name.isSimilar('Book',t) and isAbstract=**false** and ... (type: Class)
name.isSimilar('Cover',t) and isAbstract=**false** and ... (type: Class)

All of them have the same context, that is by default their container – the package *data* in our example. The condition for the context is:

name.isSimilar('data',t) and visibility =VisibilityKind::**public** (type: Package)

The resulting condition after merging the three individual conditions is:

isAbstract=**false** and isActive=**false** and ... (type: Class)

So the remaining conditions are the ones that all unmerged symbolic references have in common, only the differing condition *name.isSimilar(..)* was removed. The context remains the same since it does not differ for the unmerged changes. If necessary, the upper bound for the merged symbolic reference is increased to the number of unmerged changes (3 in this example): If *Unbounding References* has already been applied, then the merged symbolic reference keeps the bounds *[1..*]*, otherwise it is set to *[1..3]*.

The symbolic reference of this merged change matches indeed all classes *Article*, *Book*, and *Cover*, as intended, but also *Author*. In other words, merging changes might produce false positives in the matching phase. If the bounds are *[1..3]*, then the four matched elements are not a valid set because they are too many; the refinement step 5 (cf. Sect. 5.2) must again resolve that issue.

5.7.4 Grouping Model Changes: θ_G

A large set of unstructured model changes might be confusing to the user. To the best of our knowledge, all related work reflect the hierarchical structure of the compared model also in the model changes. Since model-independent differences are applicable to other than the originally compared models, we propose a different grouping strategy:
Whenever two changes affect common model elements, they are in the same group.
The realization of the grouping algorithm is straightforward:
groups are defined by connected components in a bipartite graph in which changes and referenced model elements are disjunct sets of nodes and model element references (symbolic references) are edges.
This heuristic ensures also that changes which depend on each other (cf. rules in Sect. 5.3), are placed into the same group; the reason is that dependencies occur only on changes that refer to common model elements.

The main drawback of this heuristic algorithm are again false positives: independently performed modifications may relate to the same set of model elements and, hence, result in the same group. Changes #2 and #5 in Tab. 5.1, for example, would both result in the same group because they both affect the class *Book* in model M_A. If we ignore all other changes for a moment and assume that the user performed only these two changes, they are put into the same group although the individual changes do not necessarily correlate (false positive). Grouping the entire set of changes #1–8, on the other hand, yields two groups: (#1–7) and (#8). In this case, the heuristic algorithm works as expected, because all changes representing the refactoring are placed into one group, and the remaining change is placed into another one.

5.7.5 Reversing Model Changes: θ_R

Sometimes it is useful to undo changes in a model, in other words, to apply model changes in the opposite direction. For instance, if design decisions have been made and realized in the model but are rejected afterwards; then the

model changes must be undone. The metamodel for model-independent differences describes model changes declaratively, however, it denotes the direction of model changes in the change types (for instance, *IndepAddElementChange*). Given a delta computed from two versions of a model M_A: $\Delta_a = \theta_1(\delta(M_A, M'_A), M_A, M'_A)$ (from steps 1 and 2 in Sect. 5.2), it can be used to restore M'_A from M_A and Δ_a: $M'_A = \pi(\Delta_a, \phi(\Delta_a, M_A), M_A)$, but not vice versa. That is, it is not intended to restore M_A from M'_A.

There are two obvious ways to allow undoing changes in models. The first is to adjust the algorithms and tooling to interpret the model changes in the opposite direction (additions are interpreted as deletions, etc). This is, however, an enormous effort. The second way is to reverse the model changes, (to turn additions into deletions, etc). Since all required information is available, such a reversal can easily be provided as another transformation.

In essence, all information about the unchanged and changed state of model changes must be swapped: additions become deletions, the old and new parents of movements and attribute changes are swapped, etc. The most important swap is that of corresponding elements with resulting elements, which denote the set of changed model elements in the unchanged and changed model versions, respectively. Let this transformation be θ_R. Having that done, the reversed delta can be used to restore M_A from M'_A. This means, in particular, that the following properties hold (assuming $\Delta_a = \theta_1(\delta(M_A, M'_A), M_A, M'_A)$):

$\Delta_a = \theta_R(\theta_R(\Delta_a))$ (inversion)

$M_A = \pi(\theta_R(\Delta_a), \phi(\theta_R(\Delta_a), M'_A), M'_A)$ (reverse application)

We do not provide formal proofs for these properties; however, the prototypic implementation of these concepts has proven the correctness of these properties for all tested models (see Sect. 7.6 for details).

5.7.6 Example

The model changes from Tab. 5.1 on page 36 were not applicable to model M_B in Fig. 5.2. Executing scope expansion, unbounding references, merging changes, and also grouping them yields a new set of changes that is listed in Tab. 5.12. The result of scope expansion is denoted with wildcards, e.g. **data**; the result of merging changes is reflected in columns '#' and 'old #'; the result of grouping is reflected in two rows, showing groups #1'–3' and #4'. The result of unbounding references is not visible here. The next section explains the application of model changes to another model like model M_B in the example.

#	old #	Change type	Change description
1'	1	add class	*IdElement* to package **data**, incl. attribute *id : String*
2'	2–4	delete attribute	**id* : String* from all classes (that have one) in **data**
3'	5–7	add generalization	from classes in **data** to *IdElement* (added by #1')
4'	8	update attribute	*Title : String* in *Book* to *title*

Table 5.12: List of generalized changes (Δ_s) after scope expansion, merging changes, and grouping; #1'–3' are in one group, #4' is in another group.

5.7.7 Classification of Generalization

One must keep in mind that the semantics of model changes may change when executing transformations. The matching phase (step 4 in Sect. 5.2) may, in particular, include false positives – that is, more elements are matched than intended. As a consequence, the user has to correct that manually in the refinement phase (step 5). We analyzed modifications of symbolic references and identified four categories:

Refactoring: the semantics remain unchanged; the symbolic reference matches exactly the same set of elements as before the modification.

Widening: the semantics change; the symbolic reference matches more model elements than before the modification.

Reducing: the semantics change; the symbolic reference matches less model elements than before the modification.

Altering: the semantics change; the symbolic reference matches a set of model elements that includes other model elements than before the modification, but not all of them.

Grouping changes θ_G is a refactoring transformation because it does not change the applicability to the target model. All three generalizations (θ_S, θ_U, and θ_M)[10] shown earlier are widening modifications, because the resulting symbolic references match to more model elements than before the generalization, depending on the target model. It depends on the intended scenario which type of generalization is appropriate. Using model differences in the sense of traditional patches would only allow refactoring modifications, whereas widening modifications are, for instance, useful to describe model refactorings. In most cases we would like to avoid altering modifications unless a change should explicitly be applied to another model context.

[10] Reversing model changes θ_R is non-competitive because it manipulates the change types and not their symbolic references.

5.7.8 Conclusion and Discussion

The presented transformations are used to add specific properties to a set of calculated model-independent differences. Some of them (scope expansion θ_S and merging changes θ_M) require condition-based symbolic references. The idea of these three transformations (including unbounding references θ_U) is to raise the abstraction level of model-independent differences and to make them applicable to other models. Grouping θ_G and reversing model changes θ_R, in contrast, structure the changes and reverse their direction of application, respectively.

We identified four categories of modifying symbolic references that specify where model changes are applied. *Refactoring* transformations do not alter the semantics of model changes; for example, grouping changes structures the changes without affecting their applicability. *Widening* transformations are the actual generalizations like scope expansion, merging changes, and unbounding references. The other two categories, *reducing* and *altering*, should be avoided unless the resulting changes should explicitly be applied to another model context.

All generalizations have been developed with focus on structural models like UML component or class diagrams, or Ecore models. Other models, especially when describing behavioral aspects, may require different heuristics, such as business process models [GKLE10]. This is why the framework for model-independent differences is modular and different heuristics may be provided as additional transformations.

5.8 Applying Model-independent Differences

This section covers the application of model changes to a target model, namely the matching of model elements, refinement of the mapping, and the application of model changes to the target model. There are several aspects that differ from most related work: the matching algorithm is *distributed* over all symbolic references; model changes are applicable *multiple* times; an interactive and automated *selection* of the affected model elements is possible; and already *applied* changes are detected.

The application of model changes is also called *merging models* in the literature [Men02]. The term originates from the use of versioning systems when the remote and the local version of a model are merged. However, in a standalone manner, i.e., without a common ancestor of the compared model and the model to which the changes are applied, the situation is a bit different. If the changes

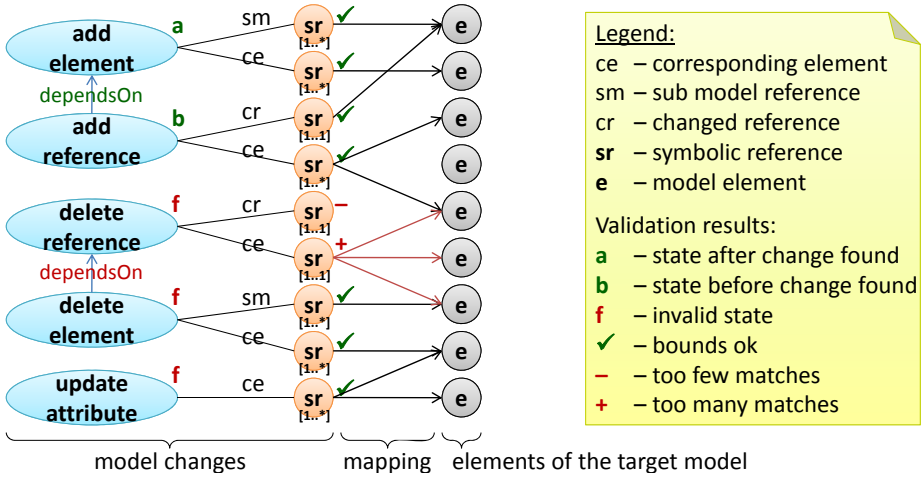


Figure 5.21: The initial mapping Ψ produced in the matching phase; only 2/5 changes validate successfully.

exist apart from any model and are then applied to another model, there are just the two artifacts, the changes and the target model. Then it is not reasonable anymore to call it merging, but *application* of changes.

5.8.1 Matching Model Elements: ψ_1

The first step towards the application of model changes is to find the locations where the changes shall be applied (matching). The matching ψ_1 produces a mapping Ψ from all changes in the delta Δ_s to the model elements of a model M_B that shall be modified: $\Psi = \psi_1(\Delta_s, M_B)$. This initial mapping Ψ is calculated automatically by calling *IElementReference.match(..)* on all symbolic references of all changes. Hence, the matching algorithm is distributed over all symbolic references. To be more precise, the symbolic references of all changes are mapped to elements in model M_B. Fig. 5.21 sketches an example mapping: five changes contain in total nine symbolic references which refer to eight out of nine elements in the target model. As indicated by the status '**a**', '**b**', and '**f**', three of the changes are not applicable with the initial mapping; the validation in the following subsection explains why.

The mapping is defined in terms of the metamodel in Fig. 5.22. It consists of *Change Mapping*s, one for each change; if the attribute *ignore* is set to true,

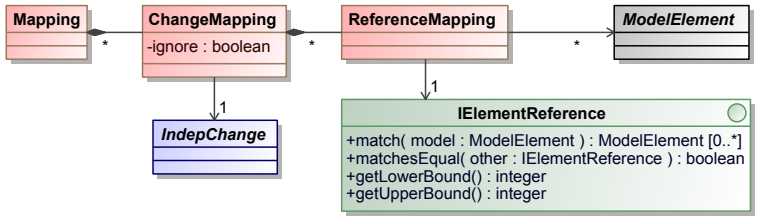


Figure 5.22: The metamodel for mappings from model changes and their symbolic references to model elements.

then this change is ignored for validation and application (in Fig. 5.21, none of the changes is ignored). A *Reference Mapping* maps each symbolic reference of a particular change to the set of matched model elements. The example mapping in Fig. 5.21 contains five change mappings (one for each change; not visible in the figure), nine reference mappings (one for each symbolic reference; also not visible in the figure), and in total twelve references to model elements (mapping arrows in the figure). Before the application of model changes can be performed, a successful *validation* of the mapping is required.

5.8.2 Validating the Mapping

The initial mapping might not be *valid*, that is, the model changes might not be applicable to the target model. The example mapping in Fig. 5.21 shows all three cases in which a change is invalid and not applicable to a model. Each of them is explained next.

1. **Bounds.** The number of matched elements must be within the bounds specified by the symbolic reference (violated by *delete reference* change).
2. **Application Status.** All matched model elements must reflect the state before the change (*add reference* change) or after the change (*add element* change), to ensure its applicability or to ensure that it is already applied, respectively (violated by *update attribute* change).
3. **Dependencies.** All changes on which the current change depends on must be valid and not be ignored (violated by *delete element* change).

1. Bounds. The bounds of symbolic references are a strict limitation of how many model elements may be mapped. Too few (marked with "–") or too many (marked with "+") mapped model elements denote an invalid mapping for the

respective change. The mapping can be corrected be adjusting the number of mapped model elements to conform the specified bounds.

2. Application Status. The mapped model elements must reflect the state before or after the change. The first change *add element* in Fig. 5.21 is mapped to two model elements, one being the corresponding element and one being a model element that matches the respective model descriptor. Hence, the added model element does already exist and the change is valid. During the application phase, no element will be added; however, a successful validation is required to successfully apply other changes that depend on the current one, like *add reference* in the depicted example. This is why we decided to detected already applied changes in the model.

Next, we discuss the state of an example change: the *add element* change in Fig. 5.21 describes the adding of a new model element. The state before the change requires the *parent* to be mapped and that the newly added element is not found as a child of that parent; to this end, model descriptors must provide the operation *isDescriptorFor(..)* (cf. metamodel for model descriptors in Fig. 5.13 on page 63). On the other hand, if any of the parent's children is described by the model descriptor, then the state after the change is found in the model. In case the parent is not mapped in the model, the change is invalid. The criteria for the states before and after a change are summarized in Tables 5.13 and 5.14, respectively. For add element changes, for example, it lists the just mentioned criteria.

The criteria for removed model elements, marked with * in the table, is different from the corresponding criteria for added elements. We do not require that the removed model element (including sub elements) *equals* the deleted element in the target model for two reasons. First, especially the applicability to other models would be very limited if that exact model element must exist, including all sub elements and properties; that would decrease the applicability a lot compared to just using the symbolic reference for identifying the model element(s) to delete. Second, the user should be able during the mapping refinement in step 5 to add additional elements that shall be deleted; if these elements have to equal the original deleted element, it is not possible to add arbitrary elements to this set. This would be an impractical limitation of that kind of change.

Detecting the state before a change is an indicator that the model change is applicable. Detecting the state after a change denotes that it will not be applied later, but other changes, that depend on the former, can still be applicable. In contrast, if a change for which the state after that change is detected, is ignored, then all other changes, that depend on the former, will also be ignored.

	Element	Reference	Attribute
Add	• Parent mapped • No element mapped that conforms to the model descriptor	• Element mapped • Reference target mapped and it is not referenced, **or** • Element target could not be mapped	• Element mapped • Element's attribute does not contain the added value
Remove	• Parent mapped • Element mapped that shall be deleted *	• Element mapped • Reference target mapped • Element does refer to that target	• Element mapped • Element's attribute does contain the added value
Update	• Moved element mapped • Parent before the change is mapped and contains moved element	• Element mapped • Reference target before the change mapped • Element refers to that target	• Element mapped • Element's attribute has the value before the change

Table 5.13: Criteria for detecting whether mapped model elements reflect the state *before* the change (all listed criteria have to be fulfilled, it not stated otherwise).

	Element	Reference	Attribute
Add	• Parent mapped • Element mapped that conforms to the model descriptor	• Element mapped • Reference target mapped • Element does refer to that target	• Element mapped • Element's attribute does contain the added value
Remove	• Parent mapped • No element mapped that conforms to the model descriptor	• Element mapped • Reference target mapped and it is not referenced, **or** • Element target could not be mapped	• Element mapped • Element's attribute does not contain the added value
Update	• Moved element mapped • Parent after the change is mapped and contains moved element	• Element mapped • Reference target after the change mapped • Element refers to that target	• Element mapped • Element's attribute has the value after the change

Table 5.14: Criteria for detecting whether mapped model elements reflect the state *after* the change (all listed criteria have to be fulfilled, if not stated otherwise)

3. Dependencies. The last validation criteria is that all dependencies must be valid and not ignored. Although the state before the change of *delete element* was found in the example model in Fig. 5.21, a depending change is invalid – hence, this change is not applicable either.

5.8.3 Refining the Mapping: ψ_2

In case the initial mapping is not valid, all invalid changes must be resolved. Both user-interactive as well as automatic resolutions, are possible. If the initial mapping is already valid, this step is optional but still the user may review and refine the mapping.

The resolution of an invalid mapping Ψ to a valid mapping Ψ', denoted as $\Psi' = \psi_2(\Psi)$, consists of two actions: (1) adding and removing model elements to/from *Reference Mapping*s (cf. mapping metamodel in Fig. 5.22); (2) ignoring entire model changes. These actions can either be performed manually by the developer or by an algorithm. The following descriptions apply to a user-interactive refinement strategy.

If too few or too many model elements are matched for a symbolic reference such that the change is not applicable, the user may add/remove model elements to/from the mapping until the number matches the denoted bounds and the change is applicable (that is, the mapped model elements reflect the state before or after the change). The developer can, in particular, also add arbitrary model elements to the mapping even if they were not initially matched by the symbolic references; the only condition is that the mapping is still valid (the bounds match and the state before or after the change is reflected in the model).

Lastly, the user can *ignore* a change. This will remove the ignored and all depending changes (which are not applicable without the ignored change) from the mapping. Ignored changes are neither validated nor applied.

5.8.4 Applying Model Changes: π

The final step in the process of applying model-independent differences takes the target model M_B, the changes Δ_s, and the successfully validated mapping Ψ' and applies all non-ignored changes to the model: $M'_B = \pi(\Delta_s, \Psi', M_B)$. The application of model changes is completely automated and works as follows (the order is important because of the reasons given in parentheses).

1. All reference deletions are performed
   (must be performed before element deletions because targets of reference deletions may be deleted).
2. All element deletions are performed
   (must be performed before element additions because a deleted element may block the location of a newly added element).
3. All element additions are performed
   (must be performed before reference additions because added elements may be targets for new references).
4. All reference additions are performed.
5. All attribute changes and element movements are performed.

This algorithm or variants are also found in existing work; nevertheless, it is mentioned here for the sake of completeness. After performing all changes in the target model, a summary is available showing the result of the application. We require such a summary because a metamodel may contain constraints on the model. Hence, the resulting model may be syntactically or semantically incorrect although each atomic model change was correct on its own. Having a log of all automatically performed changes available helps developers to analyze the situation in such a case.

5.8.5 Example

The initial matching of the generalized changes listed in Tab. 5.12 on page 80 on the model M_B (shown in Fig. 5.2 on page 37) is sketched in Fig. 5.23. The validation succeeds for changes #1'–3', but it fails for #4'; this is not surprising since there is no attribute *Title : String* in model M_B whose name could be updated. Change #1' matches one model element, the package *customerdata*; the mapping to the symbolic reference *'sm'* of change #1' is an internal model reference. Change #2' matches the two attributes *cont_id* and *inv_id*, and their respective parents. Change #3', however, matches all three classes; but the meaning of the refactoring is to match the same set of classes as change #2'.

The obvious refinement of this mapping is to ignore the invalid change #4' and to remove the false positive mapping. Then the validation succeeds and all changes can be applied to the model of change #3'. The resulting model M'_B is depicted in Fig. 5.24. A new class *IdElement* with the according inheritance relations has been added and the attributes *cont_id* and *inv_id* have been deleted.

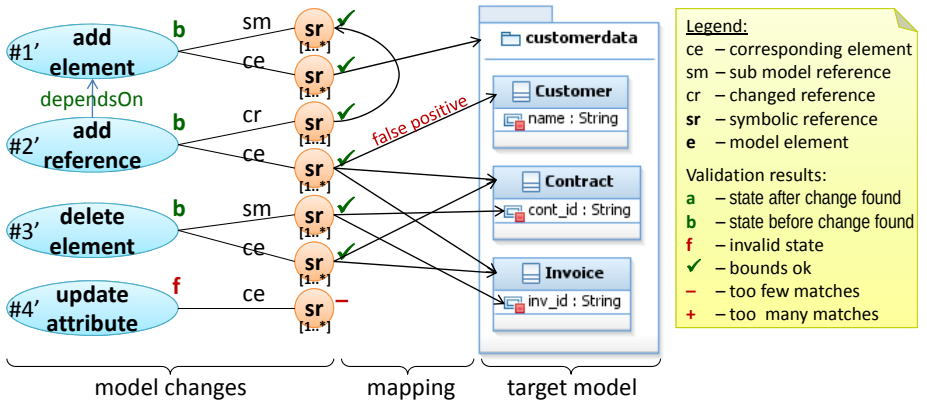


Figure 5.23: Result of initial matching of generalized changes to model M_B; the corresponding element of #4' did not match any model element, and, according to the refactoring, #3' maps one false positive.

Let us briefly discuss the false positive mapping. It originates from the generalized change #3' which was merged from three atomic changes. The generalization removed the condition for the class names, hence, all non-abstract classes are matched now. This is one of the drawbacks of generalizations – the loss of information might yield false positives. Therefore, a manual review of the mapping is highly recommended. However, one could fix that issue by manually adjusting the condition of the symbolic reference to match only classes that own an attribute that will be deleted. The following OCL expression only matches model elements that own a property whose name is similar to the string *'id'* (with a threshold $t = 0.7$). Using that condition avoids the false positive.

```
self.ownedAttribute->exists(a | a.oclAsType(Property).name.isSimilar('id', t))
```

Yet another solution would be to couple the two symbolic references denoting the corresponding elements for changes #2' and #3'. This makes sense because both shall match the same model elements. This is, however, future work.

5.8.6 Conclusion and Discussion

To summarize, the application of model-independent differences is performed in three steps: an initial matching, a refinement of the initially created mapping, and the actual application of model changes. Before any change is applied, the mapping must be validated successfully. The purpose of the refinement step is to resolve invalid mappings until all selected model changes are applicable.

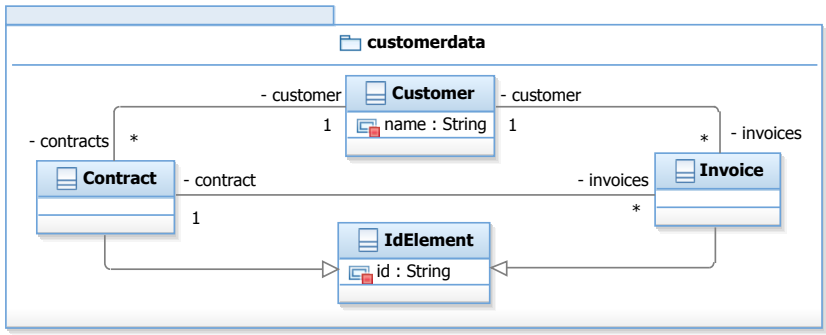


Figure 5.24: Model M_B after the application of the generalized changes.

The algorithms for validation and application are applied on a new situation in contrast to all existing algorithms for merging models or applying changes: this is the first scenario in which changes are applicable multiple times and already applied changes are detected. Only Brosch et al. [BLS+09] are also capable of applying changes multiple times (they call it *iterations*), but they are not able to detect already applied model changes.

One aspect is a high degree of flexibility to provide a framework that allows applying model differences to arbitrary models, not only the ones from which they have been calculated. Keeping each of the steps modular and exchangeable enables customizations, for instance, for specific modeling languages.

5.9 The Binding

This chapter defined the structure of model-independent differences, explained their creation, application, and several additional features like generalization to make them applicable to other than the originally compared models. These concepts are sufficient to apply model-independent differences to other models as motivated in the introduction of this chapter. When they are used for automating realizations of design decisions, however, there are additional use cases that motivate a *binding* between the individual model changes and the affected model elements.

The example in Sect. 5.8.5 maps each change to a set of affected model elements during the application of model changes. Assuming the refactoring *extract superclass* is the realization of a design decision, all changes shall remain applied in

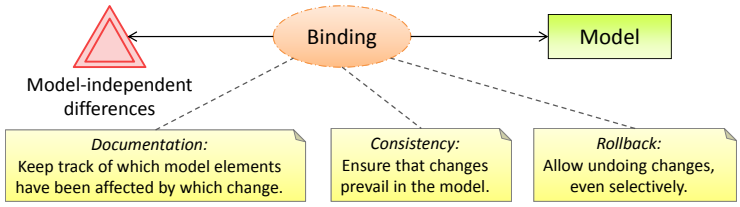


Figure 5.25: Informal sketch and purposes of the binding between model-independent differences and the model to which they have been applied.

the model. In big models and when many modifications are made, it is nearly impossible to keep track of all model modifications manually and to check whether the changes remain applied, especially if many people are working on the model. So we need some concepts for checking that automatically. Figure 5.25 summarizes the motivation for a binding: besides the just mentioned consistency, additional information about the date, author, and reason of changes provide traceability information to the developers, and rejected decisions require the possibility to undo or to roll back model changes.

The requirements for a binding are briefly discussed next, then a formal binding is defined that enables the three aforementioned use cases.

5.9.1 Requirements

To conclude the state of the art analysis from Sect. 3.3, a common format and definition of traceability relations is still missing. Drivalos et al. [DKPF08] propose an approach to automatically generate case-specific traceability metamodels, however, we need exactly one case-specific traceability metamodel. Tracing design artifacts to documents and vice versa is typically done via bi-directional relations. Concerning the use of traceability links during runtime, their manual creation is, in general, impractical and rarely beneficial. The same applies to a binding between changes and model, it should automatically or at least semi-automatically be created to be beneficial. For long-term use, literature recommends tool-supported methods for the evolution of traceability relations.

A naive binding could be a persistent version of the mapping like the one in Fig. 5.23, which is a by-product of the application process for model changes. This is, however, not sufficient for the motivating examples especially with respect to model evolution. The following requirements for a binding are derived from the aforementioned use cases.

A The binding must be created automatically. Additional effort for creating a binding is impractical and not beneficial, as discussed before.

B The binding must be easily maintainable. The model evolves over time and may affect model elements that are linked by a binding. It must be possible to edit and update the binding.

C An automated validation of all applied changes must be possible. The binding must be formally defined such that automated validation is possible.

D Information about changes must be available at the model. This information includes the date, author, and reason why the changes have been applied. Other developers must be able to easily retrieve this information.

5.9.2 The Binding Definition

The starting point for a binding could be the mapping that is used during the application of model-independent differences (cf. mapping metamodel in Fig. 5.22 on page 83). However, this mapping does neither provide a way to add additional information, nor does it differentiate between different model elements mapped to the same symbolic reference. Nevertheless, it is available as an automatically created by-product during the application of model-independent differences.

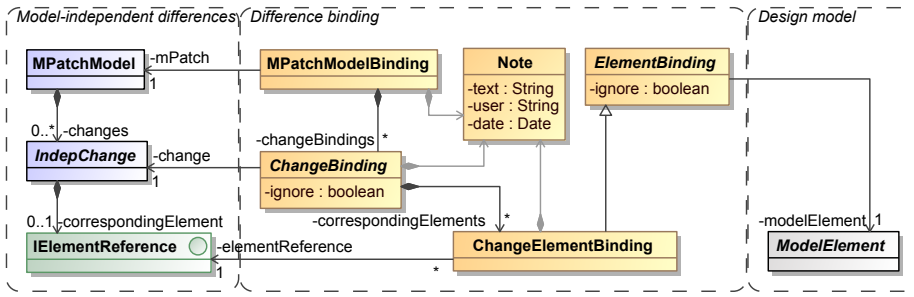


Figure 5.26: An excerpt of the metamodel of the binding between model changes (*IndepChange*) and model elements.

Figure 5.26 is an excerpt of our binding definition in terms of its metamodel showing the relation between changes and model elements. The full binding metamodel is available in Appendix C. The *MPatchModelBinding* contains the individual *Change Bindings* for all applied model changes (*IndepChange*). The latter include a fine-grained binding per model element (*ElementBinding*), that is, each affected model element has a corresponding element binding – in contrast

to the mapping, where a reference mapping links to a *set* of model elements. Moreover, individual model element bindings and also the entire change binding can be *ignore*d for validation. Additional information about the change can be provided via *Note*s.

With the proposed binding, requirement D is met because individual documentation can be added via *Notes*. Requirement B, the maintainability of the binding, is met because *Element Bindings* can be updated according to modifications in the model. Binding updates are discussed in detail later in Sect. 6.5. The automated creation of a binding (requirement A) must be realized during the application of model changes – it is straightforward to convert a mapping into a binding, so we omit the details here. The automated validation (requirement C), however, requires additional information about the change itself; for instance, which elements are affected besides the *corresponding elements*? Subtypes of *Change Binding* add this information per change type. The concrete metamodel for the binding of add reference change types is explained next.

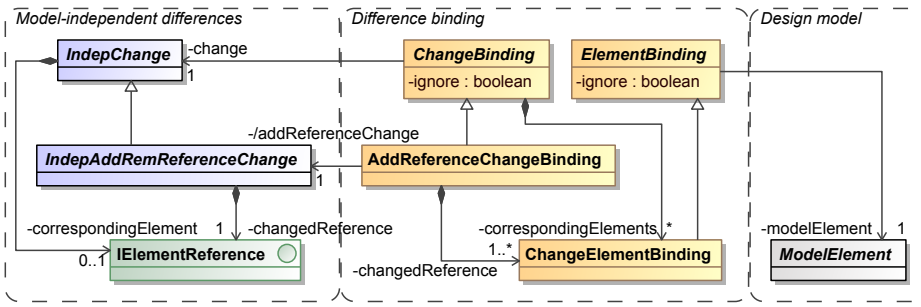


Figure 5.27: An excerpt of the metamodel for the binding concerning add reference model changes.

Besides the binding of the *corresponding element*, which denotes the model elements to which a reference was added, this change type also refers to the target model element of the reference via *changedReference* (cf. metamodel in Fig. 5.9 on page 52). Figure 5.27 shows the definition of the binding of added references which contains an additional *Element Binding* for precisely the target of the added reference. This way, all affected model elements are covered by the binding. The same applies to all other change types as listed in Appendix C.

5.9.3 Example

The following example binding of change #3' in Fig. 5.28 was created from the mapping in Fig. 5.23 on page 88. It shows how the binding links the add reference change, a UML generalization which was applied twice, to all affected model elements. Corresponding elements are the two classes *Contract* and *Invoice*, the target of the reference is the class *IdElement*.

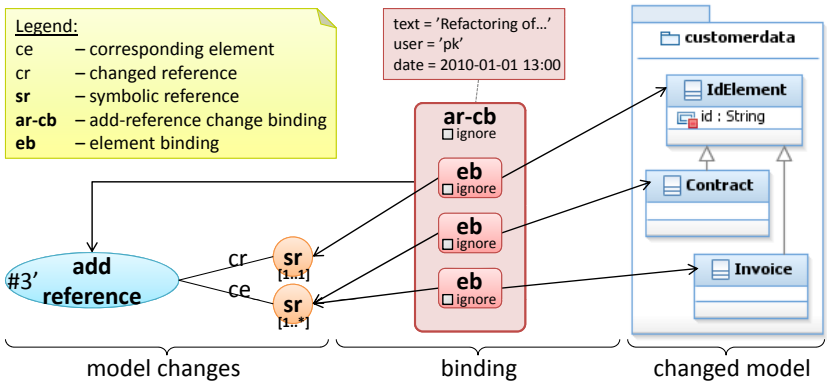


Figure 5.28: A binding for an add reference change, applied to two model elements.

All requirements are met with such a binding. A: the binding can be created automatically from the mapping. B: since the binding is specified for each affected model element, updates can be made as we will see later in Sect. 6.5. C: all affected model elements are linked via a formal binding, so constraints can be defined to automatically validate that all changes remain applied in the model (constraints are explained later in Sect. 6.5). D: additional information is available and can be added in a fine-grained manner, because notes can also be added to element bindings.

5.9.4 Conclusion and Discussion

The binding defined in this section is based on the mapping that is created during the application of model-independent differences. However, additional properties are required compared to the mapping definition: the binding is created automatically and includes information when, by whom, and why the model was changed. Moreover, it is possible to update the binding in case of

model evolution and an automated validation checks that all changes are still applied in the model.

The next chapter makes use of the binding as documentation and also to ensure that design decisions are and remain realized in the design models. Together with the reversal of model-independent differences, the binding can also be used to automatically reverse model changes and, thus, undo them in models. Further use of the binding includes customization of the application of model changes, but that is future work.

5.10 Related Work

There are many model differencing concepts that are similar to the conceptual solution presented in this chapter. Sect. 3.2 presented them already and compared their features with our solution. Here we first compare our work with related generic differencing technologies and then we compare some individual features.

Just like other differencing metamodels, e.g. EMF Compare, SiDiff, and AMOR, the representation of model-independent differences is fix and contains one metaclass for each change type. The approach by Cicchetti et al. [CRP07], in contrast, dynamically derives a differencing metamodel from the metamodels of the compared models. Since there is a multitude of fixed differencing metamodels, an abstraction to a family of domain specific languages could lead to a uniform and general core for defining model differences. Zschaler et al. [ZKD+09] propose a metamodeling language for domain specific languages which could be used to define such a family. The diverse features of each individual differencing metamodel would be variation points in their approach. This is, however, future work.

Concerning model differencing concepts, the operation recorder of the AMOR project [BLS+09] incorporates similar concepts as model-independent differences. This is the only work that has a similar referencing strategy for model elements based on OCL conditions. However, it has a different approach for applying model changes, namely by using the merger of a third-party tool (EMF Compare). Our solution, in contrast, uses a new way of applying model changes that allows to detect and bind changes that are already applied.

Although other related work is also able to transfer model differences from one model to another (e.g. [CRP07, Kol09, Eys09]), they are not capable of generalizing model changes to make them applicable in another context.

In contrast to generic differencing approaches, there are other technologies which perform specific tasks better, for instance, refactorings [RSA10] and model migration [RHW+10]. The reason is that these approaches are specifically designed for the tasks mentioned, but, on the other hand, are not as flexible as model-independent differences (supporting multiple difference calculation components, referencing and matching strategies, generalization and application algorithms, and grouping strategies). The EWL [KPRP07] also provides a flexible way for defining custom and parameterizable model changes that can be applied with user interaction; however, all change specifications must be coded manually in particular languages and cannot be specified by example models.

There are several novel features of model-independent differences that are, to the best of our knowledge, not covered in related work. The detection of already applied changes is essential in many model transformation technologies, in particular for incremental transformations [CH06], but existing model differencing technologies do not yet support that feature. The generalization of model differences by merging similar model changes into single generalized ones has been proven to be a useful feature for specifying refactorings. Since model changes are extracted and abstracted from the original model, they are applicable to other models in different contexts. AMOR is the only related work that is also capable of generalizing model differences, by manually adjusting the OCL conditions of model element references.

There is again no related work that uses the same grouping strategy for model changes. However, the experiences with refactorings and design patterns as model changes (cf. Sect. 8.1.2) as well as the domain level test (cf. Sect. 8.2) have shown that this grouping strategy works well.

5.11 Summary

This chapter presented novel concepts for a flexible representation of model differences and their use. The concepts complement existing differencing technologies that provide difference calculation algorithms. Our main use case is the creation of model differences from an example model and their application to other models; in contrast to most related work, these two models do not need to have a common ancestor. The concepts support any MOF-based modeling language such as the UML, BPMN, or custom domain-specific modeling languages based on MOF.

The key features of model-independent differences are their flexibility and modularity. Several properties can be configured to make the model differences ap-

plicable to different scenarios, for instance, to use them as model refactorings, to store and apply frequently used model fragments, or to patch other models in the traditional sense. Further, all components such as the matching strategy, conflict resolution, difference creation or application are exchangeable modules in case additional, for example, model-specific features should be added.

Symbolic references are the most important concept that allow different ways of describing which model elements are affected when applying model-independent differences to a model. They allow different matching strategies such as id-based and condition-based matching. The latter can be used to generalize model differences in order to make them applicable to models with different contents than the originally compared models.

Although model-independent differences are the basis for integrating design decisions with design models (this is what the subsequent chapter is about), the concepts are a contribution on their own right. The technology is a generic tool for describing and applying model differences and has been appreciated by the modeling community: the implementation is contributed to the open source Eclipse project *EMF Compare* and available in the Eclipse Modeling Edition.

Chapter 6

Design Decisions in Model-based Software Development

Decisions are made all the time during the development of software systems; the choice of programming languages, databases, components' architectures, but also the use of design patterns are examples. Sometimes developers are aware of these decisions, especially if they affect several components of the design. Sometimes they are not and, thus, follow their intuition: they change the design like they did in the past without considering other, maybe better suited solutions. The consequence is that the software design is hard to understand and, thus, hard to maintain and sometimes better solutions could have been chosen. We motivate an integration of design decisions as first-class artifacts in model-based software development and explain how they can be used to improve software documentation, to automate recurring work on the design, and to validate whether design artifacts conform to made design decisions.

This chapter is organized as follows. Sect. 6.1 motivates and states the goals of the integration of design decisions and design models with a continuation of the example from Sect. 2. Sect. 6.2 defines design decisions, discusses how they are used and supported today, and outlines our conceptual contributions. Sect. 6.3 and 6.4 recapitulate decision capturing, making, and realization techniques and

explains adjustments for model-based software development. Sect. 6.5 introduces novel concepts for validating consistency between design models and design decisions. Sect. 6.6 discusses how consistency could be maintained in case of design model and design decision evolution. Sect. 6.7 gives ideas for a guidance tool of design decisions in model-based software development. Sect. 6.8 gives an overview of related work and Sect. 6.9 concludes this chapter.

6.1 Motivation and Goals

We motivate our goals with a continuation of the example in Sect. 2. It deals with the development of a server application for a fictitious company E that sells devices for measuring energy consumption. Bad design documentation and collaborative work accidentally introduce inconsistencies between made design decisions and design models as explained below.

The system under development maintains customer data and values measured by devices which are installed at E's customers. During the development of the server application, a design issue *Session Management* was identified. The leading architect decided that the *Server Session State* alternative shall be realized in the design model, because there are not many clients expected and sessions can easily be handled on the server (cf. justification in Tab. 2.3 on page 14). Then she explains important characteristics concerning the realization of this decision to another developer who is then responsible for realizing the chosen alternative in the design model. The resulting design model is shown in Fig. 2.3 on page 15. The architect also documents the decision (the information in Tab. 2.3) in a decision management system and in the design documentation.

Later in the development, another developer cleans up the model and moves the session object *ContractSession* from the package *presentation* to *data* which already contains all other session objects – depicted in Fig. 6.1. Suppose, he also accidentally deletes the association from *ContractController* to *ContractSession* because he is not aware of the decision *SessionManagement*. As a consequence, this modification introduces an error in the system because parts of the realization of decision *Session Management* is now missing in the model. If he would have known about the decision, this error may not have been introduced.

Inconsistencies between design documentation (including design decisions) and design artifacts are a big problem in software development. Inconsistencies between these artifacts may produce errors in the system that are not immediately visible. The later the errors are found, the more expensive it will be to fix them [Par94]. Later, we will see how a proper documentation of design decisions

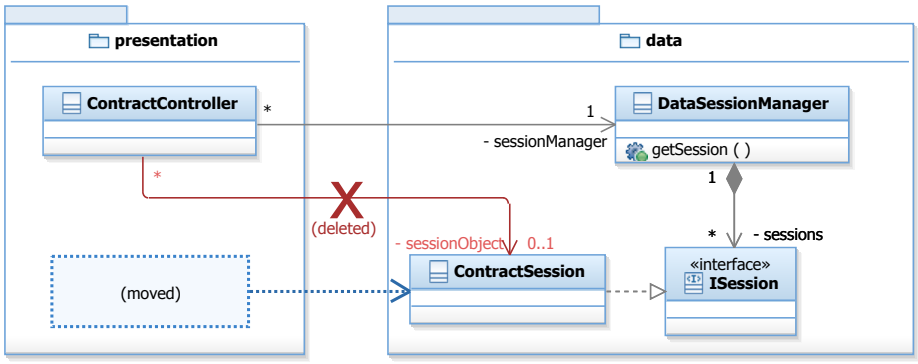


Figure 6.1: The design model has been evolved: *ContractSession* has been moved and the association *sessionObject* has accidentally been deleted.

automatically creates a fine-grained relation between design models and design decisions that can be used to automatically detect such inconsistencies.

Below, we briefly recapitulate the thesis goals and relate them to the aforementioned example.

Goal 1: improve design decision documentation. Easy to use opportunities shall be offered to developers for adding/retrieving documentation of design decisions to/from design models – instead of having the documentation separately in a decision management system or design documentation. The design knowledge shall be linked to design artifacts so that tool (and context) switching to a decision management system becomes obsolete. Thus, the design should be easier to understand.

Concerning the example, developers shall easily see the design decision (e.g. *Session Management*) which affected particular model elements (e.g. the class *ContractSession*).

Goal 2: automate recurring work. The realization of a design decision may appear in the same or in other projects. Recurring realizations are recurring work that is tedious and error-prone, especially if realizations imply many modifications in design models. The user shall be supported by the modeling tool to automate recurring model changes.

Concerning the example in Fig. 2.1 on page 12, there are at least three session

objects in the design model that are similar and managed by the same session manager; their realization shall be automated to save manual modeling effort, to avoid modeling errors, and to pave the way for uniform realizations of the same design issues.

Goal 3: consistency between design models and design decisions. An automated validation shall check whether design models still conform to all made design decisions. This way, modifications in the design which violate previously made design decisions shall be identified immediately and the user shall be notified.

Concerning the example, developers shall be notified about the inconsistency after doing the refactoring, because the deleted association was part of the realization of the previous design decision *Session Management* (cf. Fig. 6.1).

The remainder of this chapter presents definitions and new concepts for meeting these goals. A tool implementing these concepts should then support developers to avoid the aforementioned problems in the motivating example.

6.2 Design Decisions in Software Development

In this section, we discuss how and by whom design decisions are made and realized in the literature and in practice. We define design decisions and point out the causes that lead to bad design documentation and inconsistencies between design decisions and design models. Then we outline our conceptual solution as improvements for the tasks of handling design decisions.

Design decisions are made on different levels of abstraction: higher-level design decisions that also classify as architectural decisions, are typically made by software architects or groups of developers. Lower-level design decisions are typically made and realized by single or small groups of developers. Experience[1] shows that architectural design decisions are often, but not always documented whereas lower-level decisions are rarely, but at least sometimes documented in practice. The documentation could be text documents, tables, but also wikis or decision management systems.

We observed significant differences between the two roles *architect* and *modeler* that are also highlighted in Fig 6.2: architects document decisions directly in

[1]Sources are literature [KLvV06, BDLvV09] and discussions with experts (Sect. 8.3).

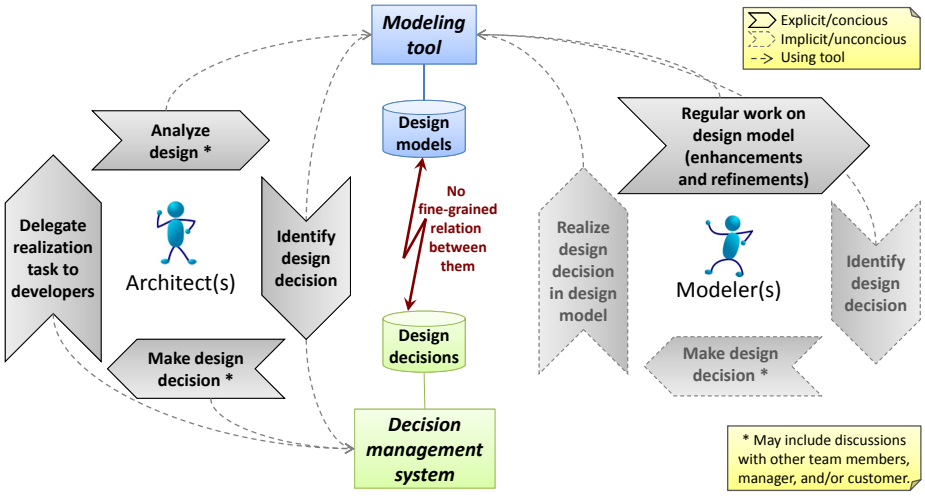


Figure 6.2: The task of making a decision depends on the role and the context, e.g. on the project's documentation guidelines or whether a decision management system is used.

design models, in a design documentation, and/or in separate decision management systems. Modelers, in contrast, either document design decisions as comments in design models and code, or not at all. Sometimes they are not even aware that they made a decision which has alternative solutions besides the one they have chosen. Hence, they make decisions implicitly and sometimes even unconsciously.

Implicit and unconscious decision making implies that decisions are not documented (tacit knowledge). Consequently, the design is hard to understand and it is almost impossible to keep it consistent with undocumented decisions, especially if many developers are working on the design. But even if decisions are documented in a decision management system or in a design documentation, as architects often do, there is no relation between design decisions and design models[2]. Hence, understanding a design and validating whether it conforms to made design decisions requires both tools, the decision management system and the modeling tool, and is tedious manual work. To overcome these problems, we propose an integration of design decisions and design models.

[2]Existing work [BB04, TJH07, JvdVAH07] provides links between decisions and design artifacts, but not on a fine-grained level for design model elements. Hence, it is not possible to check whether design models conform to made design decisions.

6.2.1 Definition of Design Decisions

This section first defines our view on design decisions based on related approaches. Afterwards we present new extensions to the definition for an integration of design decisions with design models.

There are many definitions of design decisions in the literature. An often cited definition concerning the system's architecture is made by Kroll and Kruchten:

> *"Software architecture encompasses the set of significant decisions about the organization of a software system: selection of the structural elements and their interfaces by which a system is composed, behavior as specified in collaborations among those elements, composition of these structural and behavioral elements into larger subsystems, architectural style that guides this organization."* [KK03, p. 315]

In other words, significant decisions have major impact on the architectural design of a software system by specifying its structure and behavior. Starting in 2004, Kruchten et al. published an ontology/taxonomy of design decisions [Kru04, KLvV06] which classifies four different types of design decisions (*existence decisions*, *non-existence decisions*, *property decisions*, and *executive decisions*) with properties and relations between them (more details about it in Sect. 3.1 on page 17). At the same time, Tyree and Akermann [TA05] present a decision capture template and Jansen and Bosch [JB05] even compose system architectures as sets of architectural design decisions. Zimmermann et al. [ZKL+09] focus their definition of design decisions on reuse (depicted in Fig. 3.3 on page 20).

Since the focus in this thesis is on reuse and the distinction between project-specific and project-independent parts, we adopt the definition of design decisions from Zimmermann [Zim09] which was already presented in Fig. 3.3 in Sect. 3.1 on page 20. We define design decisions in terms of the metamodel in Fig. 6.3: a design decision addresses a particular design problem (*Issue*), considering arbitrary many solutions (*Alternatives*), and contains the rationale why a particular alternative was chosen (part of the *Outcome*). Issues are further organized in *Groups* which are part of a *Decision Catalogue*. The outcome represents the *Project*-specific instance of a decision and includes assumptions and justifications whereas the issue and alternatives are independent of the actual project.

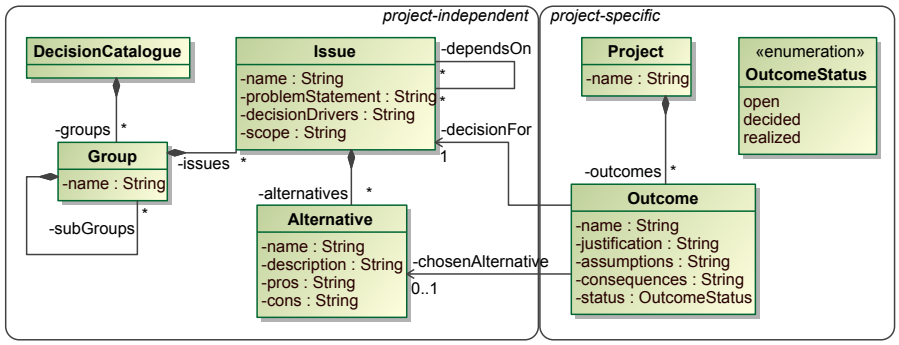


Figure 6.3: The definition of design decisions in terms of a metamodel: the decision catalogue contains project-independent issues and their solutions; the outcomes of decisions are part of particular projects.

The project-independent parts correspond to Zimmermann's reusable architectural decision model, and the complete definition (including the project-specific parts) corresponds to his architectural decision model for a particular project [Zim09, Zim11]. This results in two different models in his definition and requires an additional step to transfer design issues that are captured in a project to the reusable decision model so that they can be reused in other projects, too. We decided to avoid this redundancy of issues and alternatives in favor of their reusability in several projects without this additional step.

Like in most other approaches, design knowledge and decision rationale consist of informal information (text), specified as attributes like *problemStatement* of the issue, *pros* and *cons* of alternatives, or *justification* of decisions' outcomes. Moreover, decisions might depend on each other (*dependsOn* relation), the reference *induces* in Fig. 2.2 on page 14 is an example for such a relation. Both decisions in that figure are example instances of this metamodel.

There are several properties with this definition that are worth mentioning. First of all, the rationale of a decision, probably the most important property of related approaches in the literature, is stored in the attribute *justification* of outcomes. Second, the *status* of an outcome denotes the state of a decision: when it is set to *decided* or *realized*, the chosen alternative must be set; when it is set to *realized*, the decision should be realized in the design. Third, relations between design decisions have special meanings in the literature [KLvV06, ZKL+09]. Examples are: constrains, forbids, forces, influences, overrides, conflicts with. At this point, we do not fix the set of concrete relations but leave it open. They are discussed in Sect. 6.7.

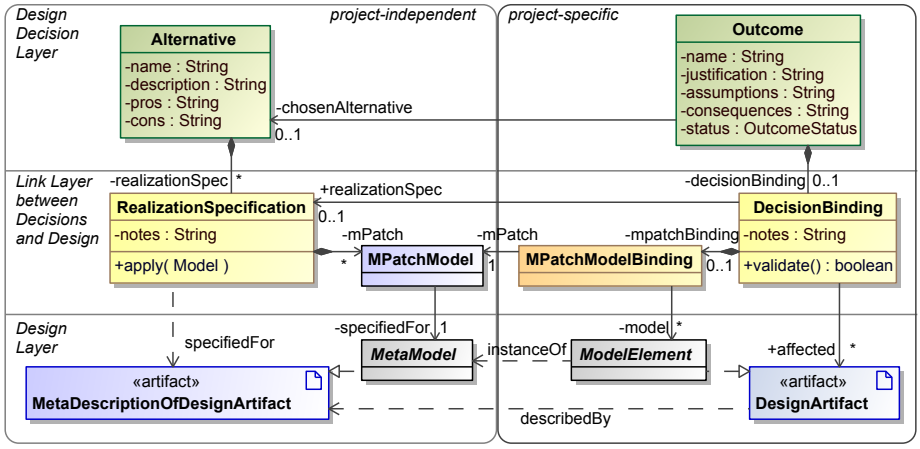


Figure 6.4: The extended definition of design decisions includes a link layer to affected design artifacts.

Extended Definition of Design Decisions. So far, design decisions are not yet connected to any design artifacts, but they are defined on their own. The status *realized* already indicates a relation between the outcome of decisions and design artifacts, but it is not explicit in most existing approaches. Kruchten et al. [KLvV06] use a *traces from* relation as a link to design artifacts, Jansen et al. [JvdVAH07] generate code out of design decisions, and Zimmermann [Zim09] triggers a model transformation on a design artifact. However, none of them provides a fine-grained link between design decisions and affected design model elements which allows, for example, precise consistency checks (goal 3).

We extend the metamodel from Fig. 6.3 with a link layer that binds outcomes to design artifacts that are affected by the respective decision (cf. Fig. 6.4). A *RealizationSpecification* is a description of how a particular alternative can be realized in a design artifact. Several realization specifications may be added to an alternative representing different variants. Design artifacts are not necessarily course-grained artifacts like an entire design model, but may also be individual design elements such as elements within a design model like classes or their attributes in UML models. A *DecisionBinding* links the outcome of a decision and the specification of the realization of a chosen alternative to affected design artifacts which realize the chosen solution. The link layer uses model-independent differences (*MPatchModel*) as realizations with a corresponding binding (*MPatchModelBinding*, cf. Sect. 5.9).

For decision *Session Management* in the example in Sect. 2.2, the binding refers to several *ModelElement*s, namely *ContractController*, *DataSessionManager*, *ISession*, and *ContractSession* as well as to the associations between them.

6.2.2 Decision Handling Scenario

This section explains a scenario of capturing, making, realizing, and validating design decisions – this is at the same time an overview of the remainder of this chapter. In the state of the art, design decisions and design artifacts are maintained separately as discussed earlier in Sect. 3.1. Our extended definition of design decisions, however, includes a link between both artifacts (cf. Sect. 6.2.1). The following scenario sketches a novel way of handling decisions that integrates both kinds of artifacts. Instead of switching back and forth between the two types of tools, decision management systems and development tools for design models, the scenario assembles all steps within one extended development tool only.

In the state of the art, strategies for making decisions involve the *identification* of a design issue, the actual *choice* of a solution that shall be realized, and finally the *realization* itself [Zim09, Tea10]. The realization is typically separated from the other activities, because design decisions and design models reside in separate tools.

The proposed scenario in Fig. 6.5 supports developers by meeting the three goals mentioned in Sect. 6.1. This scenario assumes that the proposed link between design decisions and design models exists and that a technology is available to capture and apply realizations of design decisions (model-independent differences, for instance). Each activity in the scenario suggests a use case that requires tool support: a tag 'D' denotes work on design decisions, a tag 'M' work on design models; the tags 'G1'–'G3' denote which of the goals are addressed by new or improved activities. The respective sections dealing with an activity are stated in parentheses below the activity.

Next, we go through the scenario with the help of the example. Whenever a design issue like *Session Management* is identified by a developer (activity 1a in Fig. 6.5), the issue is either known and already stored in a decision management system, or not. If it is a new issue, the developer has to specify the new design issue in the decision management system, ideally including potential solutions (1b), like *Server Session State* and *Client Session State*. As soon as the issue is available in the decision management system, the actual decision can be made, the rationale behind it can be documented, and a solution can be chosen (2a); in the example, an outcome is created pointing to the chosen alternative *Server*

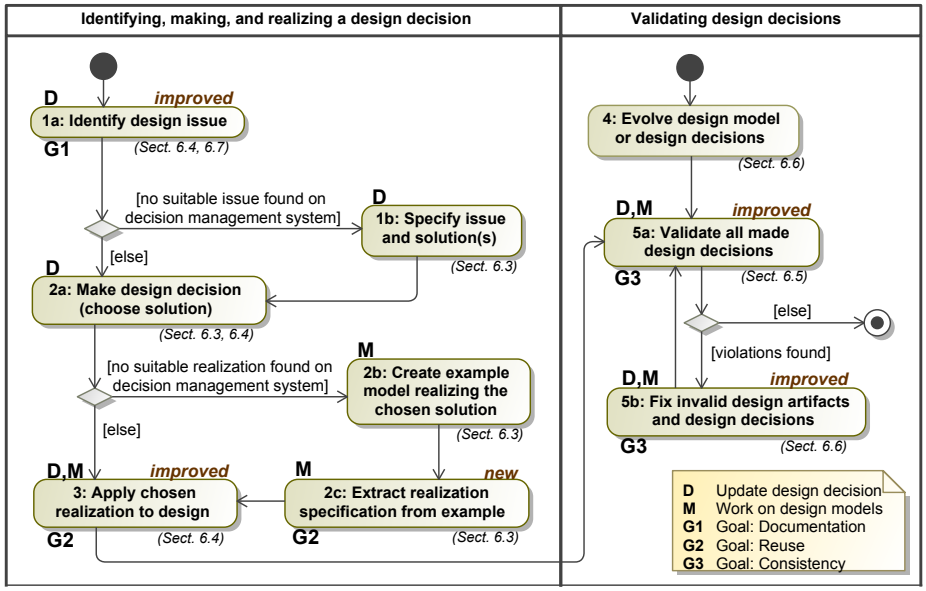


Figure 6.5: A scenario for handling a design decision: identification, making, realization, and validation; the validation comprises all made decisions and, thus, can be triggered separately.

Session State and the state of the outcome was set to *decided*. When it comes to the realization of the chosen solution in a design model, the developer must switch to the modeling tool. Realization specifications for that solution might exist, for instance, from previous projects within the same or a similar domain. In the example, the realization specification consists of several model changes, describing a new class *ContractSession* and appropriate associations. As stated previously, if a realization specification does not yet exist, developers may capture one from a design model of the project or by creating an exemplary model that incorporates a realization of the chosen solution (2b). Then the model differencing technology captures a realization specification that is applicable to other models (2c). Now a developer can apply the realization to the design model semi-automatically, that is, by manually specifying where in the model the realization shall be integrated (3). The model is then automatically modified instead of manually modeling the solution in the design. The decision's outcome status is changed to *realized*. The result of these steps for our example is shown in Fig. 2.3 on page 15. The activities in this scenario address and support goals 1 and 2: the rationale behind design decisions is explicitly captured and their realizations are partly automated.

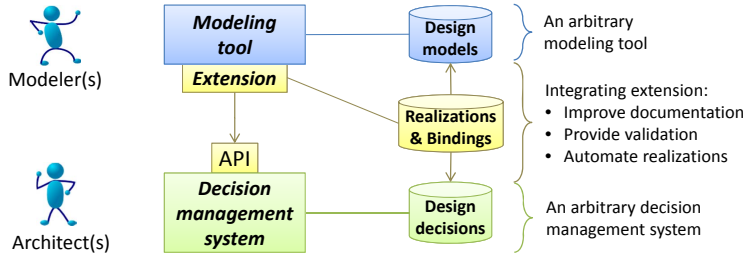


Figure 6.6: The proposed tool setup for our extension to modeling tools; decision management systems must provide an API for accessing the design decisions; this setup makes our extension independent of concrete tools.

Finally, after the decision has been realized in a design model, the current and all previously made design decisions are validated (5a). The validation is an activity that is executable at any time, for instance, whenever manual work has been done on design models (4). The validation criteria cover, amongst others, that all model elements affected by made design decisions still exist in the design models. When validating the decision *Session Management* against the corresponding design model, for instance the one in Fig. 6.1, all three classes and the associations between them must exist[3]. If that is not the case, the violating elements must be fixed (4b). Validation supports goal 3: consistency.

This scenario covers a typical life cycle of a design decision starting with its creation until its successful realization in a design model. Every time both artifacts are involved (an activity that has both tags 'D' and 'M'), developers had to swap between the two tools – this was our motivation for improving the individual tasks. The activities of identifying, making, and realizing decisions are already discussed in detail in the literature [Zim09, JvdVAH07, FCKK01, BDLvV09]. A specification of realizations that are automatically applicable to design models, and also an automated validation of made design decisions, however, are novel features that can smoothly be integrated into the design decision life cycle. Developers do not have to switch back and forth between the two types of tools anymore when performing these tasks.

The conceptual tool setup is independent of any concrete tool and is sketched in Fig. 6.6. The extension to modeling tools provides required user interfaces, for instance, for visualizing and realizing design decisions. It is in particular possible to use different decision management systems. To this end, the latter

[3]Existence of design model elements is only one criteria among others; Sect. 6.5 discusses all validation criteria.

must provide an API[4] for accessing design decisions; this interface definition is one of our contributions and is discussed in Sect. 7.5. Realizations and bindings of the link layer (cf. extended decision metamodel in Fig. 6.4) are maintained by the extension. This setup keeps the conceptual solution for realizations and bindings independent of concrete tools.

6.2.3 Conclusion and Discussion

This section points out how software architects and modelers handle design decisions, gives an extended definition of design decisions, and presents a typical scenario of a design decision life cycle. We improved several tasks to support the three goals stated Sect. 6.1: (1) design documentation is improved by supporting developers in documenting decisions and linking them to design models; (2) automated realization of design decisions eases recurring and error-prone manual modeling work; (3) a validation of made design decisions identifies inconsistencies between both artifacts.

Our definition of design decisions is based on the one by Zimmermann and describes them as design issues, potential alternative solutions, and outcomes of the decisions [Zim09, Zim11]. Our extension to this definition is a link layer to design models. This includes realization specifications of alternatives and a fine-grained binding to individual design artifacts such as design model elements.

The decision handling scenario covers the identification, capturing, making, realization, and validation of design decisions. Further decision-related activities like editing or rejecting decisions are not listed here to preserve a clear overview of a decision life cycle. However, our concepts and the tool implementing the concepts support such additional activities.

The proposed binding between design decisions and design artifacts allows yet more use cases than the ones discussed so far. One example is a proposal of subsequent design decisions exploiting the relation between design decisions. Sect. 6.7 discusses this feature.

[4] *Application Programming Interface*; an interface that enables a program to interact with other programs.

6.3 Capturing Design Decisions

Before design decisions can be made, they should be captured. Capturing denotes the documentation of design issues together with potential alternatives, typically in a decision management system, without making any decision. To clarify, decision capturing is a project-independent activity to prepare the project-specific activities of making and realizing decisions. This section first discusses the state of the art in decision capturing before it presents our solution.

6.3.1 State of the Art in Decision Capturing

There are many concepts in the literature for capturing architectural knowledge including design decisions [uhB09, LJA10, TAJ+10, LK08a]. The three methods below are suited for capturing reusable design decisions in model-based software development.

Prospective decision capturing (also known as elicitation of design decisions) is a dedicated decision modeling step in which domain experts prepare design issues and alternative solutions and document them, for instance, in a decision management system [Zim09, BGJ05]. This extra effort pays off when making decisions, because only a choice must be made, documented, and the chosen solution must be realized; the issue itself and potential solutions are already documented. Moreover, this way gives other developers the opportunity to extend their design knowledge when they have to make or realize decisions.

Retrospective decision capturing (also known as bottom-up capture), in contrast, is the documentation of already made decisions [CND07]. For instance, during dedicated decision capturing phases after a design or parts of it are finished. The advantage is that there is no documentation effort before and during the decision making and realization activities, only afterwards. However, making decisions requires information about design issues and potential solutions, which takes a lot of time – prospective decision capturing does that beforehand. Also, developers must recall their decisions and justifications afterwards; decisions, potential solutions, and rationale may have been forgotten and, thus, could be missing in the documentation.

On-the-fly or *ad-hoc* decision capturing (also known as top-down capture) documents design issues, selected alternatives, and captures the realization specification *while* making or realizing decisions. Hence, the documentation is integrated into the activities of making and realizing decisions. This method also allows partial decision capturing pro- or retrospectively.

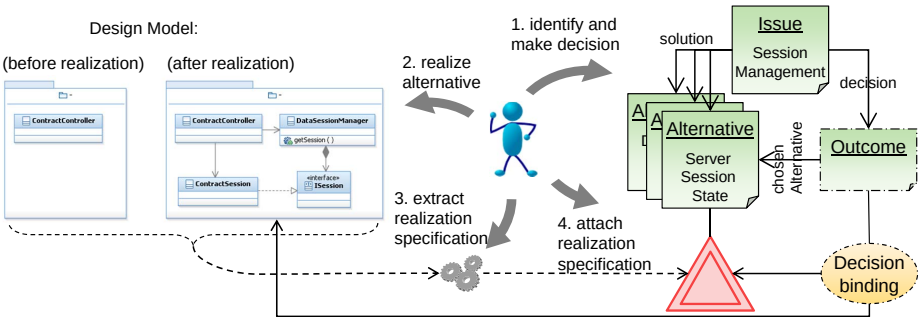


Figure 6.7: Ad-hoc decision capturing also includes to identify, make, and realize a decision; issue, alternative, and outcome are created ad-hoc, if required.

We experienced ad-hoc decision capturing mostly with software modelers and rarely with software architects. The latter typically capture decisions prospectively or retrospectively. Our solution supports all three methods; ad-hoc decision capturing is explained in more detail using the running example.

6.3.2 Example

Suppose a developer captures the decision *Session Management* from Sect. 2.2 on page 13 ad-hoc (sketched in Fig. 6.7). First, she identifies the design issue *Session Management*, creates an outcome, and makes the decision to realize the alternative *Server Session State*. Second, since there is no appropriate realization specification available for this alternative, she realizes it in the design model manually. Third, she uses the technology for model-independent differences to extract the realization specification and, fourth, attaches it to the alternative. The third step includes the creation of a decision binding (cf. Sect. 5.9).

Since the developer used the ad-hoc capturing method, she also *made* and *realized* the decision. Both activities are explained in detail in Sect. 6.4.

6.3.3 Extended Decision Capturing Methods

This section presents our conceptual solution as an extension of the three existing decision capturing methods. When capturing design decisions, one or many realization specifications can be attached to alternatives. These realizations can be extracted from exemplary models (in case of prospective decision capturing)

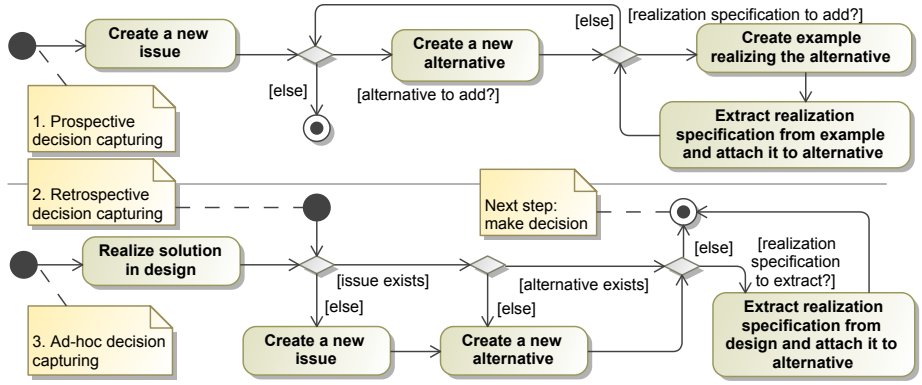


Figure 6.8: The three extended methods for decision capturings: prospective decision capturing is independent of any project whereas the other two methods require a concrete design artifact.

or from existing project models (either ad-hoc or retrospective decision capturing). Later, when making a decision for which a realization specification is attached, developers are able to semi-automatically apply this realization to a design model (cf. application of model-independent differences in Sect. 5.8). All three extended decision capturing methods are shown in Fig. 6.8 and work as follows. They assume that a developer already identified the decision to capture.

1. *Prospective decision capturing* is independent of any project. It describes the creation of new issues, arbitrary many alternatives, and for each alternative arbitrary many realization specifications. A realization specification can be extracted from exemplary models and attached to an alternative.
2. *Retrospective decision capturing* requires a concrete project. It describes the creation of an issue that appeared during the design of a project; if such an issue is already documented, the existing issue can be used, otherwise a new issue is created. Then an alternative is created that was chosen for the selected issue; again, if such an alternative exists, the existing alternative can be used. If the decision's realization in the design models can be expressed with model-independent differences, developers can extract a realization specification. The subsequent step is to document the decision they just made (next section).

3. *Ad-hoc decision capturing* is the same method as retrospective decision capturing, but during and not after the design phase. As presented in the example in Sect. 6.3.2, decision identification, the realization of the

alternative in design models, and decision capturing are tightly integrated. Again, the subsequent step is to make the decision, explained in the next section.

In contrast to the existing capturing methods in the literature, the extended methods include realization specifications of alternatives. This is possible because model-independent differences can be used to specify changes in design models. In case a realization cannot be specified as design model changes or a decision does not imply a realization, the activities concerning realizations can be skipped. Otherwise the methods integrate the creation of realization specifications from examples into the decision capturing activity. Since examples in design models are usually available in retrospective or ad-hoc decision capturing, the additional effort is very low. In prospective decision capturing, the effort is only low if example models already exist.

For capturing realization specifications, the additional effort is the configuration of model-independent differences and adding additional information (*information* of realization, cf. extended definition in Fig. 6.4). This should include information about the author, the date of creation, as well as a brief description including further references about that particular realization, for example, a reference to a design pattern definition.

6.3.4 Conclusion and Discussion

The existing decision capturing methods in the literature do not comprise realizations in the design. Three methods dominate: prospective, retrospective, and ad-hoc decision capturing. Each of them is reasonable depending on the use case scenario, so we support all of them for meeting our goals 1 and 2 (improved documentation and reusing realizations).

The challenge is to distill realization specifications without much extra effort, and that they are applicable to other models when realizing the decision. We use model-independent differences to describe realization specifications, because they can be created from exemplary models and generalizations make them applicable to arbitrary models (cf. Sect. 5.7). Then we extended existing capturing methods with new tasks that integrate these realizations into the decision capturing activities.

6.4 Identifying, Making, and Realizing Design Decisions

The two tasks of identifying and making a decision are closely related to each other and some people see them as a single step. The identification of a decision occurs whenever a developer becomes aware of a design issue in a concrete project. Making the decision, however, denotes the selection of a solution for an identified issue. The final step is its realization in the design. Although the order of these three steps is compulsory, the time frame is open – there might be days or weeks between the identification of a decision until its realization. All three steps are discussed in this section, first the state of the art, then problems concerning model-based software development, and lastly our solution.

In the example introduced in Sect. 2, the identification of the design issue *Session Management* could be made in several ways. One could be that a developer working on the design model recalls from personal experiences that a controller in a web application should use and maintain a session. Another way would be that a developer reads about this issue in a decision management system and recognizes the issue during the design of a system. Part of the identification step is the documentation of the particular issue and the creation of an open outcome[5]. *Making* the decision, on the other hand, is the choice of the alternative *Server Session State*; the outcome status is then updated to *decided*. After the chosen alternative has been realized in the design as illustrated in Fig. 2.3 on page 15, the outcome status is set to *realized*.

6.4.1 Decision Identification

There are many activities in the literature that involve the three abovementioned ones. Tang et al. [TAJ+10] present activities that do not only cover reusable design decisions (part of *general knowledge*), but also context, reasoning, and design knowledge. However, we spcificaly concentrate on the identification of design decisions, list the state of the art, highlight problems in practice, and discuss ways to support developers with this task.

The *identification* of design issues must be made by a developer and, thus, the identified issues depend on the developer's skills and knowledge. Architects are typically more practiced in this step than modelers as discussed in Sect. 6.2. Design studies, software documentation, reading technical and research papers, and

[5] An open outcome is an instance of *Outcome* having the status *open*, cf. metamodel in Fig. 6.4 on page 104.

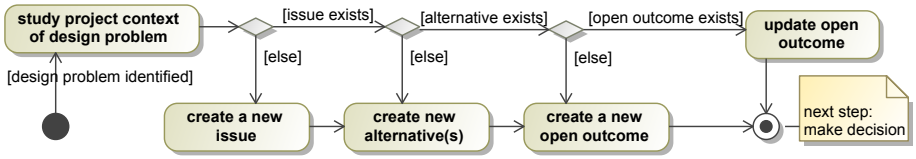


Figure 6.9: Prepare the documentation of an identified design issue: create issue, alternative, and open outcome, if necessary.

especially personal experience are the main resources for being able to identify design issues [FCKK01, Zim09, TAJ+10, BGJ05, Tea10]. The method of Zimmermann proposes an explicit technique for decision identification: systematic analysis of available pattern documentation and available sources for architectural knowledge lead to stepwise identification of design issues in a project.

Problem. Learning about potential design issues, i.e. studying design knowledge resources, takes a lot of time. As we recall from personal experience and from experience of other developers (also reflected in the interviews in Sect. 8.3), the chosen solution for a decision is often the one that is already known from previous projects. The reason is that developers feel confident with previously chosen solutions and it does not take extra time learning about a new solution. The drawback is that potentially better suited solutions are not considered and the rationale for the design might not be objective. Even worse, since software development is typically a time-critical undertaking, developers tend to neglect documentation tasks, but rather concentrate on the realization and implementation only. In the end, the design rationale is only "in the developers' mind" which causes not or badly documented design knowledge to vanish eventually. Furthermore, a realized solution in the design may not be the best suited one, because alternatives were not considered.

Solution. The idea is to make it as easy as possible for developers to access available design knowledge in order to minimize the effort for learning about alternative solutions, and to tightly integrate the decision identification task into the development environment. Therefore, we recommend the method of Zimmermann [Zim09] and briefly outline in Fig. 6.9 how to properly document an identified design issue in a project.

Suppose a developer identified a design issue either from personal experience or from another knowledge source, the next step is to study the issue in the context of the current project. A decision is then prepared in a structured way in terms

of an *issue* to document the general design problem; one or many *alternatives* denote potential solutions; and an *outcome* denotes the actual decision that has to be made. Since the decision is not yet made, the outcome status is *open*. The last four activities – creating/updating issue, alternative(s), and outcome – are explicitly supported by the development tool to avoid a context switch to another tool. In regard to the tool setup in Fig. 6.6, these activities are provided by the modeling tool extension.

6.4.2 Decision Making

The decision making step in our solution does, just like the decision identification step, not differ from techniques in existing work. Therefore, we also briefly discuss the state of the art of this step as it can be used with our concepts.

Making a design decision requires that developers are aware of the design issue for which a decision must be made. Alternative solutions must be available or may be captured ad-hoc (cf. decision capturing in Sect. 6.3). The result of making a decision is the selection of one of the available alternatives together with a justification.

Falessi et al. [FCKK01] analyse existing decision making techniques. They consider alternative properties such as terminology (e.g. "*performance*") and measures (e.g. in seconds) and evaluate potential alternatives based on different weighing strategies, fulfillment estimations, and uncertainty factors. Each of the considered techniques can potentially be used with our conceptual solution.

In our solution, the task of making a decision is supported by both tools, the decision management system and the extended modeling tool, in order to offer the functionality to both roles, software architects and modelers (cf. tool setup in Sect. 6.2). This avoids the need for switching tools. The subsequent task of realizing design decisions is discussed next.

6.4.3 Decision Realization

This section explains how a design decision is realized in design models after it has been identified and made. However, not all decisions can be realized in the design. Other than existence decisions (cf. design decision ontology in Sect. 6.2.1) do not necessarily relate to particular design elements. Because of this, the concepts in this section are only applicable to decisions, whose realizations are expressible in terms of design model changes! As discussed in

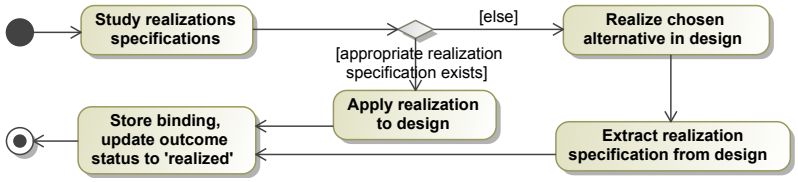


Figure 6.10: Realizing a design decision: either an existing realization for a chosen solution can be used or a new realization can be created.

Sect. 3.1, there is, to the best of our knowledge, no approach that supports design decision realizations in design models.

Problem. In contrast to code, realizations of design decisions in a model-based software development context cannot be described that easily. The two fields, model differencing and model transformations, are potentially capable of detecting and describing changes in design models. In Sect. 5.1 on page 38, we already discussed the pros and cons of both approaches. We decided to use model-independent differences to specify realizations of design decisions. Besides the technical challenges, another challenge is to properly integrate realizations in the design process. That is, realization specifications must be easily producible and applicable by all developers with as little extra time and effort as possible.

Solution. An outline of our solution is given in Fig. 6.10. First, a developer studies existing realization specifications, maybe there is already one available that suits the current design model. If so, the developer is supported by the tool to apply that realization to the design. Otherwise, the developer may realize the chosen alternative in the design and, with the help of model-independent differences, the realization can be extracted as a realization specification and attached to the chosen alternative. This way, the new realization specification may be reused whenever the same alternative of this decision is chosen again – in the same or another project. Lastly, the outcome status is set to *realized* and all changed design model elements are bound to the particular realization specification. The concepts for the binding between alternatives, realizations, and the design have been defined in the metamodel in Fig. 6.4 on page 104. The binding can later be used to validate whether decisions remain applied in the design – Sect. 6.5 explains that in detail.

The idea behind this solution is to avoid redundant work, especially recurring modeling tasks in design models. Realizations can be extracted and generalized as realization specifications with model-independent differences (details in

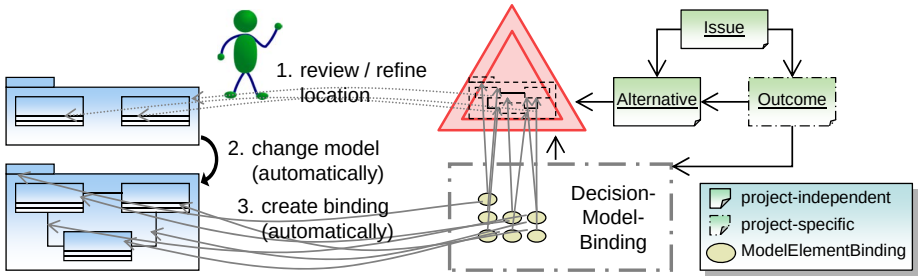


Figure 6.11: Using model-independent differences as a realization specification in three steps; the model is changed and a binding is created automatically.

Sect. 5.7) which makes them applicable to other models and, thus, reusable. Hence, if a realization specification has been attached to an alternative, it can easily be reused the next time this particular alternative has been chosen.

6.4.4 Example

Let us continue the example from Sect. 6.4.2. Assuming that a developer made the decision to realize the alternative *Server Session State* in the design model, the tool applies the realization specification semi-automatically to the design model. Figure 6.11 sketches the individual steps (recapitulation of Sect. 5.2). First, the tool proposes an initial mapping of model elements that are about to be changed and the developer may review and refine that mapping. Second, the tool updates the model automatically according to the realization specification and the refined mapping. Third, the tool creates a binding that links all affected model elements to the outcome of the decision.

6.4.5 Decision Binding Visualization

The previous sections explained how to identify, make, and realize a design decision. The result is a properly documented decision, its realization in the design, and a binding between them. Here we discuss how the binding can be exploited for visualizing this relation.

Once the binding links design decisions and design artifacts with each other, it can be used as traceability links: the binding supplies information about which design elements are affected by design decisions. The tool integration is sketched

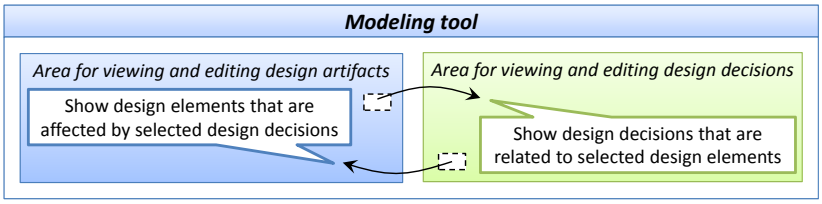


Figure 6.12: The binding can be used to navigate between decisions and design artifacts within the modeling tool.

in Fig. 6.12. It allows navigating between design decisions and design artifacts in both directions.

Whenever a set of design elements is selected, e.g. UML packages, classes, or attributes, the tool shows all related design decisions, their outcomes and rationale, but also information about the respective design issues. This is convenient for retrieving the documentation for particular design model elements. Vice versa, whenever a set of design decisions is selected, the tool shows all affected design elements. This way, the user gets a quick overview of the decisions' impact in the design.

There are two related areas in the literature, design decision visualization and traceability. Kruchten et al. [LK08b] present tool support for different visualizations, for instance, chronological or as a decision dependency graph. But this does not involve design artifacts. The traceability community, discussed in Sect. 3.3, exploits links between individual design elements and requirements or design documentation in order to provide a better understandability of the design. To the best of our knowledge, these two fields have not yet been combined on a fine-grained manner such that individual design elements can be traced to design decisions and vice versa.

6.4.6 Conclusion and Discussion

Design decisions are first identified, then made, and finally realized in the design. There are no new requirements to the decision identification and making steps, so we refer to existing work. The decision realization in design models, in contrast, is a new feature. Realization specifications are automatically extracted from design models and can semi-automatically be applied to other models. This way, redundant realization work on design models can be reduced.

Overall, our vision is to use a decision management system as a knowledge repository and ultimately as a decision guidance tool. Model-independent differences have the advantage that they can easily be created from exemplary models. There are generalization algorithms available to make model-independent differences applicable to arbitrary other models.

One must keep in mind that realization specifications must be created carefully by developers because they may be reused multiple times, also by other developers. If there is a modeling error in a realization specification, it is of course applied whenever that particular realization is chosen. Nevertheless, the task of applying realizations automatically shows its strength when realizations are rich in detail. Hereby we meet goal 2: automation of recurring work (cf. Sect. 6.1).

A critical success factor is developers' discipline to properly identify design issues, weigh up alternatives, and document made decisions – these steps cannot be enforced by a tool, it can only support them. In particular programmers but also modelers may neglect these steps and simply realize the solution which they already know because the tasks of identifying and making decisions does not appear beneficial to them. Therefore, benefits must be visible to developers and the extra effort must be minimal. Besides the automated application of realizations, the benefits are goals 1 and 3: better documentation (cf. Sect. 6.4.5) and validation (next section).

6.5 Validating Design Decisions

The previous sections explained how design decisions can be used in model-based software development to support forward engineering and to improve a system's documentation. The degree of automation can be increased for realizations of decisions and a binding between decisions and design model elements can be exploited for improving navigation between both artifacts. But how do we keep the design model consistent with made decisions? How to find out whether or not design models conform to the outcomes of all decisions?

In this section, we define formal constraints on the binding to automatically validate whether design models conform to the outcomes of design decisions. In essence, these constraints check whether any of the model elements, which are affected by made design decisions, violate a realization as it was applied to the design. If so, the user will be notified and supported to fix these violations.

Inconsistencies might introduce errors which shall be found as early as possible as discussed in Sect. 6.1. We use the example in Fig. 6.1 on page 99 to illustrate

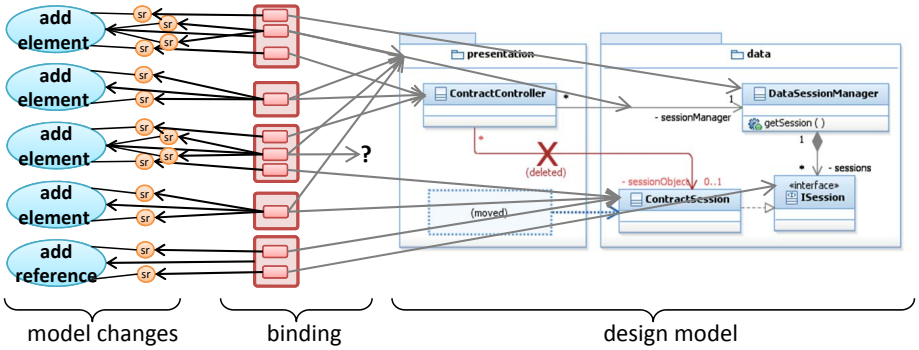


Figure 6.13: The binding for the decision *Session Management* can be used to detect modifications in the design that violate the applied realization (model changes).

the constraints and their violations. Figure 6.13 sketches the binding for the decision *Session Management* along with the evolving model: the target of one link was deleted (association between *ContractController* and *ContractSession*) and the class *ContractSession* was moved to another package. The binding is the same as introduced in Sect. 5.9. Before explaining how to automatically detect such violations, we discuss the state of the art in literature and practice.

State of the Art. There are several approaches that are similar to our validation of design decisions. Timbermacine et al. [TDSF10] present a method in which architects can manually attach constraints to architectural components in a constraint language similar to OCL. However, these constraints must be specified manually for each component. Wahler [Wah08] presents an approach to automatically propose and manually refine consistency constraints on class diagrams. But these constraints do not involve other artifacts like design documentation or design decisions. AREL [TJH07] links design decisions to design models, however, these traceability links are only used for documentation and rationale analyses, but not for consistency checks. In the end, manual peer reviews dominate in practice (cf. Sect. 8.3) which is a tedious and time-consuming task.

Solution Outline. The validation whether design models conform to made design decision must neither require manual constraint definitions nor manual reviews. The idea is that a set of formal constraints on the binding check that all applied realizations prevail in the design. In addition, developers are able to

add custom constraints on the design, e.g. using OCL. We first discuss the levels of abstraction on which constraints occur and then illustrate some excerpts for constraints on each of the levels.

6.5.1 Constraint Levels

Our goal is to ensure that all changes, which were applied during the realization of design decisions, prevail in the respective design models. Before we can check that, we must ensure that all affected design model elements exist. Altogether, there are constraints on three different levels of abstraction which are explained below.

1. *Element level: all design model elements linked to the binding must exist.*
   This level is independent of any concrete design decision and any realization specification that has been applied to the design model. It requires that bound elements must exist in the design model.
   Example: the class *ContractSession* is referenced by the binding and, thus, must exist in the design model.
2. *Realization level: all changes applied from a realization specification must prevail in the respective design models.*
   This level is specific for realizations which have been applied due to a design decision. The design must reflect all changes that have been applied as part of the design decision's realization.
   Example: if a class is changed to being abstract, that class must remain abstract.
3. *Decision level: additional custom constraints for design decisions.*
   Constraints on this level are specific for design decisions and do not necessarily relate to realization specifications. They are specified manually by the developer during design-time.
   Example: the classes *Controller* and *SessionManager* must be located in the same package in the design model.

Constraints for the first two levels are static – we defined them once and for all below. Custom constraints (decision level), in contrast, can be specified by developers during design-time and concern individual decision-related properties in design models.

Some constraints are more important than others. It is, for example, a severe violation if a design model element, that has been added as part of the decision's realization, does not exist anymore. It is, on the other hand, not that severe

if the added element has been moved to another place. Nevertheless, this is still considered a violation because it is located at another place than specified during the realization of the decision. Therefore, we classify constraints either as *warnings* or as *errors*.

Excerpts/examples for each of the three levels follow. We use the OCL (*Object Constraint Language* [OMG06b]) to define them as invariants in the following Sect. 6.5.2–6.5.4.

6.5.2 Element Level Constraints

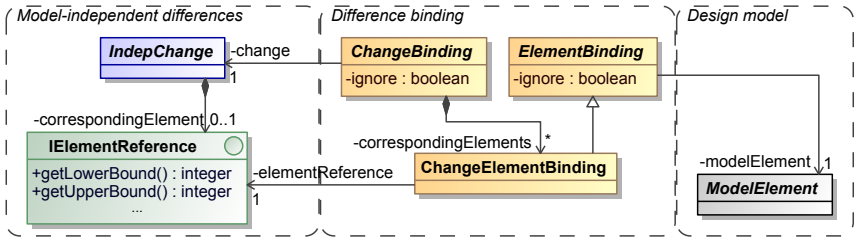
Element level constraints check whether all elements in the design exist which are affected by design decisions. Since the number of bound design elements may be restricted by the realization specification that has been applied, this number is also checked. In other words, if any bound design element does not exist anymore or the number of bound design elements differs from the specified number in the realization, the binding is violated on the element level.

The example model in Fig. 6.13 violates one of these constraints: the association between *ContractController* and *ContractSession* was bound due to the decision *Session Management*, but it has been deleted. So the developer is notified about the missing association. Then the developer has to fix this violation – this is explained later in Sect. 6.6.

The three OCL invariants in Fig. 6.14 reflect the aforementioned constraints. They apply to all *ChangeBindings* (*context* in line 1); relevant parts of the binding metamodel to understand the constraints are shown in the upper part of Fig. 6.14. The first invariant (lines 4–5) checks that all referenced design model elements are defined. The number of bound model elements is specified by *lowerBound* and *upperBound* of symbolic references (*IElementReference*) and is checked in lines 10–14. Furthermore, each of the binding elements may be *ignored* in which case the related constraints are not checked (lines 8, 11, and 13). This is important when fixing a violated binding, see Sect. 6.6 for details.

6.5.3 Realization Level Constraints

If all bound model elements exist, realization level constraints check whether the design model conforms to the applied realizations of made design decisions.




```
1  context ElementBinding
2  ----------------------
3  -- check that the referenced elements exists
4  inv Model_elements_exist:
5    self.ignore or (not self.modelElement.oclIsUndefined())
6
7  context ChangeBinding
8  ---------------------
9  -- check bounds of corresponding elements
10 inv Lower_bound_corresponding_elements:
11   self.ignore or self.correspondingElements->size() >=
12     change.correspondingElement.getLowerBound()
13 inv Upper_bound_corresponding_elements:
14   self.ignore or self.change.correspondingElement.upperBound < 0 or
15     self.correspondingElements->size() <=
16       self.change.correspondingElement.getUpperBound()
```

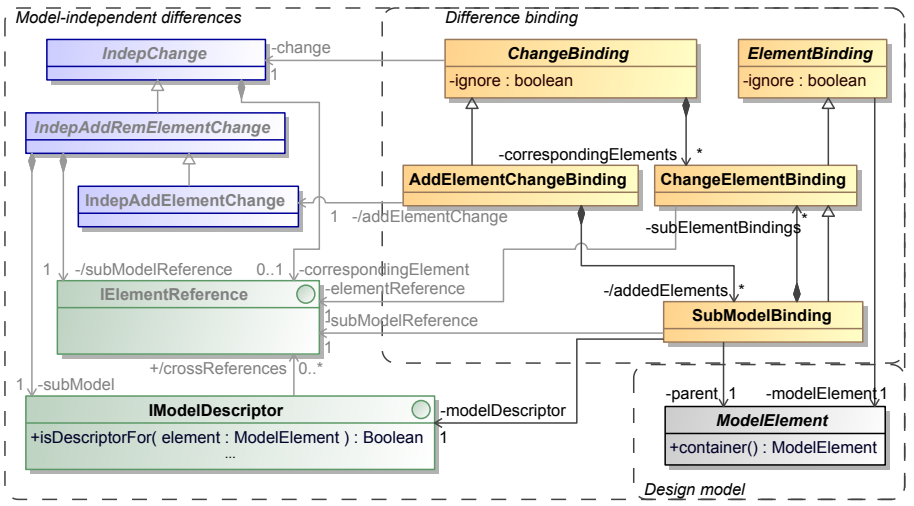
Figure 6.14: Three constraints for the element level in OCL (severity: error) and the related parts of the binding metamodel; the constraints validate the existence of model elements and that their number is within the specified bounds.

The constraints validate that all changes of the applied model-independent differences prevail in the design model. There are nine possible change types (cf. Sect. 5.3) which must be covered by the constraints. In other words, if any change, that has been applied during the realization of a decision, has been undone or modified otherwise, the binding is violated on the realization level.

The example model in Fig. 6.13 also violates one of the realization level constraints: *ContractSession* was originally added to the package *presentation* but it has been moved to *data* as shown in the figure. This is not as severe as a deleted element, so the user is notified with a warning that the design does not conform anymore to the realization of the decision *Session Management*.

The constraints in Fig. 6.15 refer to one type of model changes, namely those describing added elements; hence, the context is *AddedElementChangeBinding* (line 1). The first constraint checks whether added elements are contained in their expected containers. It uses the reflective call *container()*[6] (line 6)

[6]This constraint requires a MOF-compliant [OMG06a] metamodel because the reflective operation *container() : ModelElement* is provided by MOF.

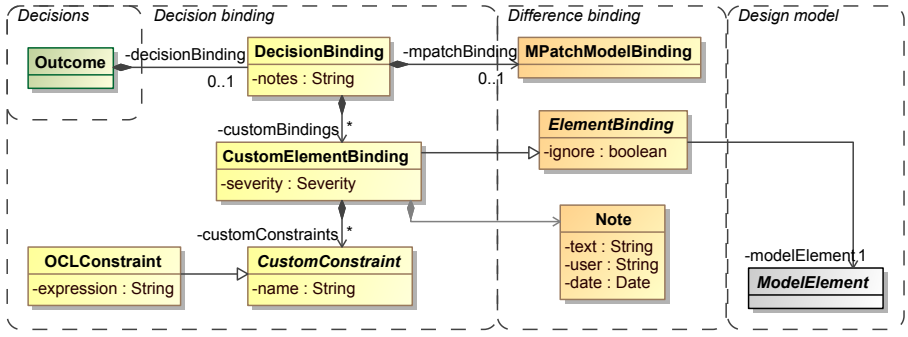



```
1  context AddElementChangeBinding
2  ------------------------------
3  -- Added model element exists at the expected parent
4  inv Added_model_element_is_child_of_correct_element:
5    self.ignore or self.addedElements->forAll(e | e.ignore or
6      e.modelElement.container() = e.parent)
7  -- Added model element must not have been modified
8  inv Added_model_element_is_not_modified:
9    self.ignore or self.addedElements->forAll(e | e.ignore or
10     e.modelDescriptor.isDescriptorFor(e.modelElement))
```

Figure 6.15: Two of the constraints for the realization level in OCL (severity: warning) and the related parts of the binding metamodel; the constraints validate that added elements have not been modified and that they are contained in their expected parents.

to retrieve the actual container and compares it with the value *SubModelBinding.parent*; the latter stores the container to which the elements have been added during the application of the change.
The second constraint checks whether the added elements have been modified. It uses *IModelDescriptor.isDescriptorFor(..)* on each of the added elements (line 10). Model descriptors must provide this operation which checks whether the given model element differs from the originally added element and, thus, has been modified. Again, the constraints are only evaluated if the flag *ignore* is not set.

The constraints for the remaining eight change types are similar to the one just presented and listed in Appendix C.




```
1 -- the context is specified via the reference to a 'ModelElement'
2 context WebApplication::presentation::Controller
3 ------------------------------------------------
4 -- parent of controller and session manager must be equal
5 inv Controller_container_equals_session_manager_container:
6   self.ownedAttribute->select(name = 'sessionManager').type.owner =
7     self.owner
```

Figure 6.16: A constraint example for the decision level in OCL (severity: warning) and an extension of the binding metamodel; the constraint checks that the element *Controller* in the design model is located in the same package as the referenced *SessionManager* (see corresponding model in Fig. 2.3 on page 15).

6.5.4 Decision Level Constraints

Decision level constraints can be specified by developers during design-time of a project at any time after a decision was made. Such custom constraints allow developers to validate additional company-, project-, or model-specific properties. Hence, in contrast to the previous two levels, custom constraints are specific for a concrete design model. Suppose the design model of interest is the one in Fig. 6.13, a custom constraint could be that the two classes *ContractController* and *SessionManager* must be located in the same package.

The upper part of Fig. 6.16 extends the decision binding definition from Fig. 6.4 on page 104 with custom constraints. *CustomElementBinding*s can be added by developers to specify a constraint for a particular model element that must be specified via the inherited reference *modelElement*. Then the developer may add arbitrary many *CustomConstraint*s, for instance, as OCL expressions.

The OCL invariant in the lower part of Fig. 6.16 is the constraint mentioned before: the *context* is the UML class *ContractController* in the *presentation* package (line 2), whose *owner* must be equal to the *owner* of the referenced *sessionManager* (line 5). *OwnedAttribute* is a UML-specific property that holds

all attributes and also refers to other model elements [OMG10]. Just like all other constraints, whenever this constraint is violated, as is the case in the illustrating example, the developer is notified.

6.5.5 Conclusion and Discussion

This section introduced concepts and constraints for an automatic validation that design models conform to made design decisions. The validation criteria are based on a binding between decisions and models which is a by-product when using model-independent differences to realize a decision in a design model. The validation is successful if all bound model elements exist in the design and they reflect the realizations that have been applied. In addition, developers may specify custom constraints during design-time. Using all these constraints, it is now possible to automatically validate that design decisions are realized in design models.

The validation does obviously only work if a binding between made decisions and the design model exists. If a design decision cannot be expressed in a realization specification or if the decision was realized manually without capturing the changes in the model, validation is not possible. Examples for decisions whose realizations cannot be expressed are other than existence decisions like coding or modeling guidelines (cf. Sect. 6.2.1). One could, however, add custom constraints for these cases like the one in Sect. 6.5.4.

A remaining question is how to fix violated bindings? This is answered in the next section which deals with the evolution of design models and decisions and its consequences.

6.6 Design Model and Decision Evolution

The design of a software system rarely remains as it is initially created. Fowler [Fow99] motivates that good software developers consistently adjust existing designs to suit the circumstances, for instance, when extending it. The main driver is to improve maintainability and extendibility of the software design. Extending and changing the design is called *design evolution* or, in terms of models, *design model evolution*. The very same happens with design decisions. They are made, rejected, extended, and updated as the system evolves. We call this *decision evolution*.

The two typical use cases concerning design model evolution are the extension of the design to add new functionality, and non-functional modifications to improve the design structure. Figure 6.1 on page 99 illustrates an attempt to improve a the structure of a design model by moving the class *ContractSession* to another package which already holds other session objects. However, an association has been deleted accidentally. These changes on the model result in two violations as explained in Sect. 6.5: one violation on the element level (missing element) and one violation on the realization level (wrong location of added element).

Since we consider design decisions as first class artifacts, we have to consider their evolution, too. Concerning project-specific parts of decisions, an important situation is the rejection of decision. If a decision has not been realized yet, there is no relation to the design. However, if a decision has already been realized in the design, then rejecting it typically implies to undo the realization. This could become an arbitrary complex step, depending on the complexity of the realization. Moreover, it might require to undo follow-up decisions.

The evolution of design decisions might also include modifications within realizations that have already been applied to the design. For example, to add model changes to realizations or to modify them.

In order to fix such violations, the developer has to identify the cause of the violation and has to decide how to correct it. In case of errors, like the deleted association, the design or the corresponding decision must be corrected to fix the violation. Otherwise, if all modifications are as intended and correct, like the moved class, then the binding must be updated and corrected. In this section, we explain the different cases for fixing violations.

6.6.1 State of the Art in Design Evolution

The problem of keeping the design consistent with other artifacts like requirements or other kinds of documentation, is well-known in the traceability community. A partly automated approach is presented by Mäder [MCH10]. It uses the observer pattern to detect modifications of traced design artifacts and then automatically updates the traces, based on a set of predefined rules. However, the definition of such rules is required beforehand and the design artifacts must be edited with a particular editor. There are other solutions for specialized settings like the one just explained, but, to the best of our knowledge, a generic approach for consistent traceability links does not exist [SZ05].

In practice, two methods dominate for findings errors and inconsistencies between decisions and the design[7]. The first are tedious and time-consuming manual design reviews. The second are regression tests. Automated test suites for the functionality of the software system are a tool for quickly identifying potential errors in the system when the design evolves. This requires, however, that proper test cases were specified [UL07].

6.6.2 Fixing Violations

In this section, we explain how the different kinds of violations can be fixed. We have the following requirements to the solution in order to provide the developer a method for easily fixing the violations:

(1) All violations must be visible to the developers at the location of cause.
(2) Developers must be able to fix violations manually.
(3) It must be possible to ignore the cause of violations for further validation.

We identified three ways to fix violations after discussing the situation with other experts and the interviews in Sect. 8.3 confirmed that they cover all practical cases.

- *Revert the modifications in the part of the design or decision that causes the violation.* This way is recommended in case of accidental modifications in the design or decision. The previous versions can, for instance, be restored from a versioning system.
- *Adjust the violated binding/custom constraint such that the constraints are met again.* This way is recommended in case the modification on the part of the design or decision causing the violation was made on purpose and the constraints shall remain active.
- *Ignore the violated binding.* This way is only recommended if the violated constraint is not relevant anymore in the current design.

So in order to fix a violation, the developer needs to identify the cause and then has to choose either of the three ways. The reasonable way to fix a particular violation may not be possible, for instance, if corrections in the models conflict with other decisions or requirements. Then the developer is responsible for resolving this issue. Each of the predefined constraints on the element and

[7] We recall these two methods from our own experience and from discussions with experts; see also Sect. 8.3.

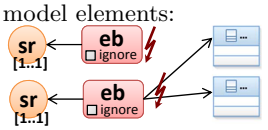
Level	Violation	Possible Fixes	Cause
Element	(A) Dangling reference: 	• ignore binding • assign new model element	*M*
	(B) Wrong number of bound model elements: 	• ignore binding • adjust bounds • assign/unassign model elements until bounds match	*M*, *D*
Realization	(C) Model change does not prevail in the design: 	• ignore binding • assign/unassign model elements until realization is restored	*M*, *D*
Decision	(D) Custom constraint is not met: 	• ignore binding • assign/unassign model elements until the constraint holds • adjust custom constraint	*M*

Table 6.1: Violations and possible ways to fix them; *Cause* indicates which evolving artifact may cause the violation, *D*ecision or *M*odel.

realization level has a brief description which helps, together with the bound model elements, to identify the cause of violation.

Table 6.1 gives an overview of all kinds of violations that can be raised by the constraints from Sect. 6.5. Column *Cause* indicates whether the cause is located in a design *D*ecision or design *M*odel. The violations are grouped into the three constraint levels.

On the element level, two types of violations may occur. In case a bound model element does not exist anymore, the binding has a dangling reference that produces a violation (A). This violation can be fixed by re-assigning a model element to this binding. If the bounds do not match the number of bound model elements (B), the bounds or the number of bound model elements were changed – either of these two causes must be corrected.

On the realization level, each constraint belongs to a change from a realization specification which was applied to the design model. Either an update of the change in the realization specification or modifications in the design model cause the violation (C). Assigning the correct model elements is the typical way of correcting the violation. In rare cases it is the realization specification that must be corrected.

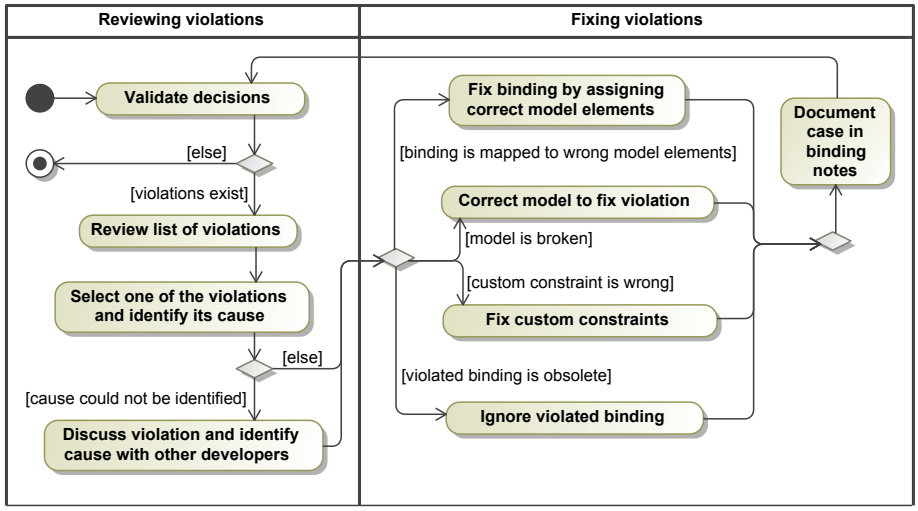


Figure 6.17: The process of how violations are shown to and fixed by developers; properly documenting the case is important for other developers to understand the rationale behind the design.

On the decision level, all constraints are provided by developers (D). Violations can either be fixed by adjusting the constraint, by adjusting the design, or by assigning the right design model elements. The choice depends on the context.

Although the process of validating design decisions and fixing the violations seems to be intuitive, we briefly outline it here to emphasize the documentation of violations, how they are fixed, and what the reasons behind them are. This information is just as important as the justification behind design decisions! Maintaining this information in a separate document or maybe in a decision management system is one possibility, but attaching it directly to the binding – and, hence, at the elements which were violated – makes it easier accessible for other developers. The definition of the decision binding already support such notes (cf. binding metamodel in Fig. 6.4 on page 104).

Figure 6.17 summarizes the process of validating design decisions and fixing potential violations. After the decisions have been validated, the violations should be reviewed one by one. Especially in collaborative software development, communication within the team might be required in order to identify the cause of a violation. If the cause is known, one or multiple of the aforementioned actions must be performed. Afterwards, the violation and its fix are documented such that other developers can follow the reasoning of the design.

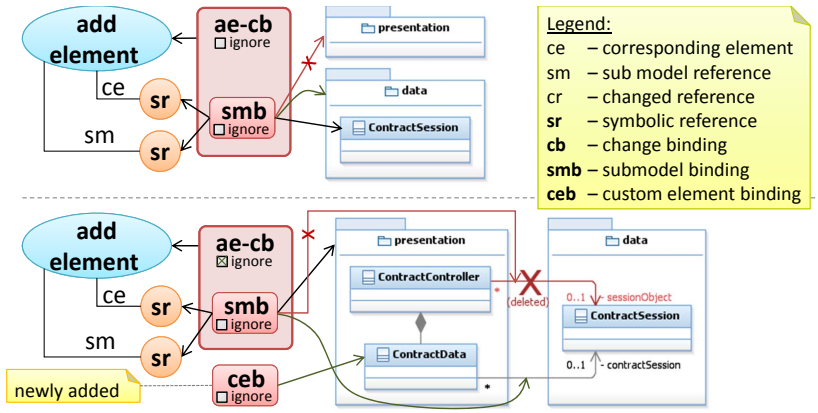


Figure 6.18: Fixing two violations: re-assign model element (2x), ignore binding (1x), add new element to binding (1x).

6.6.3 Examples

This section explains how the exemplary violations from Fig. 6.13 on page 120 and a violation of a custom constraint can be fixed. These five violations cover all cases in Tab. 6.1.

The first violation is the move of *ContractSession* to another package (case (C) in Tab. 6.1). To fix that violation, the binding can either be ignored, the model can be reverted, or the binding can be adjusted. We choose the last option, so we have to change the bound package from *presentation* to *data*, as shown in the upper part of Fig. 6.18. The binding element *smb* is an instance of *SubModelBinding*, defined in the metamodel in Fig. 6.15 on page 124. Once the binding is updated, the constraints in the lower part of Fig. 6.15 are not violated anymore.

The second violation is the deletion of the association from *ContractController* to *ContractSession* (cases (A) and (C) in Tab. 6.1). That can be fixed by replacing the deleted association with a new component *ContractData* which is now responsible for holding a reference to the session object (see lower part of Fig. 6.18). This time, fixing the binding is a bit more complex: first, we need to adjust the binding to consider the new association from *ContractData* to *ContractSession* instead of the deleted one; second, we need to disable the check whether the association is as specified in the realization (second constraint in Fig. 6.15); third, we want to make sure that the class *ContractData* exists in the model. The first part is just like the previous example. The second part requires that we ignore the validation check whether the added element is as

specified – the *ignore* flag of the add element change binding (*ae-cb*) does that. The third part requires a new custom element binding (*ceb*) which checks the existence of *ContractData* (cf. constraint for element bindings in Fig. 6.14 on page 123).

The third violation is the rejection of a decision that has already been realized in the design. The task of fixing this consists of three steps: first, if the decision has follow-up decisions that must be rejected as well, this must be done first (recursive task). Second, the realization in the design must be undone – in terms of model-independent differences, the easiest way to do this is to reverse the model changes (cf. Sect. 5.7.5 on page 78) and apply them to the model. Third, the binding of the rejected decision must be deleted.

The fourth violation is a modification in a realization specification that has already been applied to the design, for instance, lowering the bounds of one of the model changes or adding or removing a model change (cases (A), (B), or (C) in Tab. 6.1). Again, the design and the binding could be adjusted to the changed realization specification, similarly to the first two violations.

The last exemplary violation is the custom constraint in Fig. 6.16 on page 125 because *ContractController* and the session manager reside in different packages (case (D) in Tab. 6.1). Here the development team should discuss whether this constraint is required (resulting in an appropriate model refactoring) or not (ignoring or deleting this constraint).

6.6.4 Conclusion and Discussion

The design as well as the decisions of the design evolve during the development of a software system. There are plenty of ways how this happens, like manual work, automated model-transformations, or merging branches in a versioning system. If a binding exists between decisions and design models, the validation from Sect. 6.5 is able to reveal some inconsistencies between these two artifacts. These inconsistencies are violations of predefined constraints. This section discussed the different kinds of these violations and their causes. Lastly, we outlined how the violations can be fixed and how consistency can be restored.

We propose a tool-supported but to large extend manual strategy for fixing inconsistencies. The approach by Mäder et al. [MCH10] could be a useful extension because it allows to automatically update the binding in case of model refactorings. This could automate some of the re-assignments of model elements. In the end, the benefits of maintaining consistency must be higher than the effort for fixing violations, and this extension would reduce the effort.

6.7 Proposing Subsequent Design Decisions

This section presents some ideas towards a decision guidance tool to support the developer in identifying decisions. It comes without a supporting implementation or any other evidence, so it is an outlook to future research directions. With decision guidance we mean both, the proposal of subsequent design issues after a decision has been made, and a restriction of the decision space.

The example from Sect. 2.2 illustrates the proposal of a subsequent design issue (also shown in Fig. 6.19): the chosen alternative *Yes* of the first decision, *Session Awareness*, induces the follow-up design issue *Session Management*. We call this type of relations *inducing relations* because they induce other decisions. Figure 6.19 also shows another kind of relations: the alternative *Lightweight Server* of a design issue *Server Architecture* is *excluded* by the alternative *Server Session State*. We call this type of relations *restricting relations*, because they restrict the decision space.

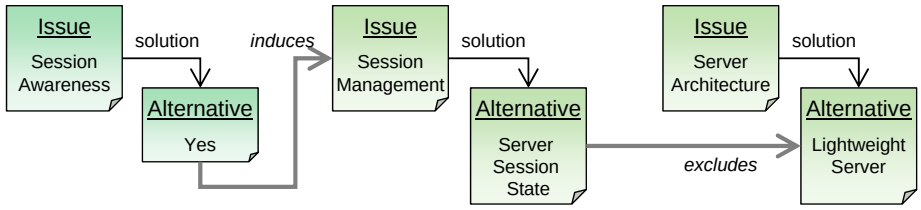


Figure 6.19: Relations between decisions in the example: *induces* is an inducing relation, *excludes* is a restricting relation.

Our hypothesis is that relations between design decisions can be used for guiding developers through the decision space by proposing subsequent and hiding incompatible decisions. This section discusses the state of the art in decision guidance, presents our conceptual ideas, and concludes with a discussion about this topic.

6.7.1 State of the Art in Decision Guidance

Current practice is to manually identify related and subsequent design issues in a decision management system, in pattern specifications, or any other source of documentation (see also Sect. 6.4.1). Although some of the existing decision management systems support relations between decisions, there is no guidance support yet as motivated. The ADK Web Tool [ZKL+09] formally defines

seven types of relations between the different parts of design decisions, listed in Tab. 6.2. They use these relations for educational purposes and for helping knowledge engineers and software architects to detect design flaws. Zimmermann [Zim11] also describes a guidance model for architectural decisions, but that concerns the identification of relevant design issues in a project that is not explicitly based on relations between decisions. The relations in Kruchten's ontology [KLvV06] do not distinguish between issues and alternatives; however, most of the relations can be expressed with the concepts of the respective other approach as shown in the table.

Ontology	ADK Web Tool	From	To	Type
isRelatedTo	isRelatedTo	Issue	Issue	inducing
subsumes	refines	Issue	Issue	inducing
comprises	decomposesInto	Issue	Issue	inducing
—	forces	Alternative	Alternative	inducing
forbids / conflictsWith	isIncompatibleWith	Alternative	Alternative	restricting
constrains	isCompatibleWith	Alternative	Alternative	—
enables	triggers	Alternative	Issue	inducing
overrides	(implicitly expressed as decision outcomes)			—
isAlternativeTo	(implicit: alternatives of the same issue)			—
isBoundTo	(implicitly expressed as groups)			—
dependsOn	—			inducing

Table 6.2: Relations between design decisions in existing approaches: Kruchten's ontology [KLvV06] and the ADK Web Tool [ZGK+07]; the latter defines relations more fine-grained, hence the columns *From* and *To*.

Recently, a tool for visualizing relations between decisions has been presented by Lee and Kruchten [LK08b]. It shows all made decisions as a graph with the nodes being decisions and edges being relations. This work also points out the difficulty to capture relations between design decisions which is tedious work, even if decisions are already captured. Possible sources may be developers' experiences, finished projects, pattern languages, etc. However, to the best of our knowledge, neither of these nor any other approach exploits such relations to propose subsequent design decisions or to filter possible choices during the decision making process.

6.7.2 Towards Decision Guidance

In this section, we present ideas for a decision guidance tool. The two types of relations are defined as follows. When using the term *decision*, it might be the issue or any of its alternatives.

1. *Inducing relations* are all relations from one decision to another which specify that the other decision may or must be made eventually.
2. *Restricting relations* are all relations from one decision to another which specify that the other decision must not be made.

Examples for both types of relations are shown in Tab. 6.2. Most relations are inducing relations, each with a particular meaning that can be derived from their names. Only one of them (*forces*) requires the induced decision to be made, all others are optional. These two types of relations can be exploited to either (1) propose upcoming design issues or (2) hide issues or alternatives that are not allowed. The motivation is that this step saves time and effort for identifying design issues and considering alternatives.

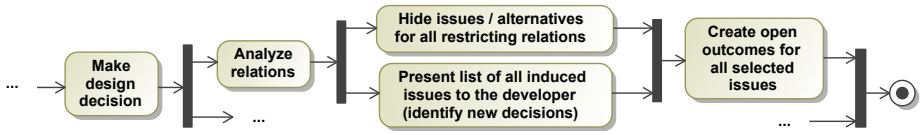


Figure 6.20: After a decision was made, inducing and restricting relations can be used to support the decision identification task (extension of the activity in Fig. 6.5 on page 106).

We propose the steps in Fig. 6.20 as an extension to the decision making activity that has been sketched in Fig. 6.5 on page 106. Whenever a decision was made, i.e. the outcome status was set to *decided*, the tool analyzes all relations of this decision. For all restricting relations, the respective issues and alternatives are hidden or marked, e.g. as limitedly applicable, in the decision management system. Further, a list of all induced design issues are shown to the developer in order to identify new design issues – new open outcomes will be created for the selected subset of these issues.

This shall, however, only be a guideline and not strict rules as the following scenario motivates. The same decision may be made multiple times in a project but for different and maybe independent components. Then, an excluding relation forbids a particular issue, but only on one component. If the decision on the other component is different, the issue that is forbidden in the first component, may very well be relevant for this other component. Hence, restricting the decision space must consider the scope of decisions in the design.

6.7.3 Conclusion and Discussion

This section explained how relations between design decisions can be exploited to ease and partly automate the decision identification step. Decisions might either be proposed via inducing relations or limited by restricting relations. Based on this information, developers can easily create new open outcomes for related issues.

As mentioned before, tools for proposing decisions can only support developers in performing their tasks, but it does not substitute thinking. This is why the proposal of issues as well as filtering alternatives should be an optional feature for developers. Then one can take advantage of that feature but still all options are possible.

6.8 Related Work

There are several approaches in the literature that are similar to the concepts presented in this chapter. All of them have already been presented and briefly explained in Sect. 3.1. In this section, we compare our solution to the most similar approaches grouped by the three goals.

Design decision documentation. Most decision management systems' main feature is the documentation of architectural knowledge including, but not limited to, design decisions. ADDSS [CND07] and PAKME [BWG05], for instance, are web-based tools; the Knowledge Architect [LJA10] is a tool suite that integrates with several office tools; Archium [JvdVAH07] and SEURAT [BB04] integrate with source code, AREL [TJH07] stores design decisions as profiled UML models; the ADK Web Tool [Zim09] and SEI wiki [BM05] are wiki-based tools for documenting and maintaining design decisions.

Although some of the approaches link design decisions to design artifacts like source code, only AREL is capable of linking them to individual design model elements. However, AREL does not allow developers to capture and reuse changes in design models, and to synchronize this information with decision decisions; these two artifacts merely coexist. None of these approaches provides a fine-grained link to individual design model elements as our solution does. Since the ADK Web Tool supports recurring design decisions with a distinction of generic and project-specific parts of decisions, we adopted its metamodel and use cases for capturing and making decisions.

Automating design decision realizations. In the past, the design decision rationale and architectural knowledge communities focused on documenting decisions that have already been made (following a retrospective capturing approach). In his architectural decision modeling framework, Zimmermann [Zim09] shows that model transformations can automatically be triggered in a model-driven development process. However, the transformations only support business processes so far and they must be specified manually. Archium, in contrast, is a framework with a notation and code generator for design decisions and source code. It can be used to compose the architecture of architectural decisions which contain realizations as source code templates.

The conceptual extension of design decisions with realization specifications and a fine-grained binding to design artifacts are novel features that none of the discussed approaches support. However, the idea of Archium to automatically create large parts of the system's design is similar to our solution which automates the realizations of design decisions in design models in a generic way.

Consistency. The consistency goal is only met for manual design constraints (decision-level constraints) so far: Tibermacine et al. [TDSF10] provide a constraint language similar to OCL that allows architects to manually attach constraints to architectural components. Generic constraints as presented for the element and change level, however, are not available.

In practice, informal, human-centric techniques such as coaching, architectural templates, and code reviews dominate. For instance, software engineering processes like the RUP (Rational Unified Process [Kru03]) advise architects to enforce decisions by refining the design in small and actionable increments. The agile community emphasizes the importance of face-to-face communication and team empowerment [Bec99]. Maturity models such as CMMI [Tea10] recommend rigid approaches to ensure that decision outcome are realized, for instance, formal reviews. Applying these techniques takes time and their success depends on the architects' coding and leadership skills.

We are not aware of model-based software development tools that respect design decisions. Modeling tools like the IBM Rational Software Architect[8] and Borland Together[9] provide pattern authoring capabilities which are similar to the intention of the realizations of design decisions. However, a metamodel for expressing relations between them as well as decision identification and decision making supported are missing.

[8] Available at: http://www.ibm.com/developerworks/rational/products/rsa/
[9] Available at: http://www.borland.com/us/products/together/

6.9 Summary

This chapter presented a new way of handling design decisions that complements existing approaches in the literature. Design decisions are a key artifact for expressing and reasoning about the design and design knowledge, and decision management systems can be used to document and maintain them. Made design decisions imply changes in the design of a system, which are in case of model-based software development, changes in the design models. The presented concepts use model-independent differences to automate recurring work on the models and to keep design models consistent with made design decisions.

Design decisions must first be identified in a particular project, captured (documented), and finally made. These tasks are widely discussed in the literature and often neglected in practice, because they take a lot of time and they rarely appear beneficial to developers. Moreover, decisions are typically maintained in separate tools or documents than the design artifacts like code or models. We close this gap by integrating the decision identification and capturing tasks into the modeling environment in model-based software development. One benefit is improved documentation, because the documentation is directly annotated to the design artifacts. After a decision has been made, its chosen solution must be realized in the design of the system.

Up to now, solutions of design decisions are typically described informally as text, with literature references, or with examples. This means that each time a decision is realized in a project's design, this task is tedious and error-prone work. We use model-independent differences to automatically specify decision realizations from examples which are then applicable to arbitrary models whenever the particular decision is made. This automates large parts of the tedious and error-prone manual realization tasks and, as a by-product, creates a binding between the made decisions and affected design model elements.

Keeping the evolving design consistent with its documentation including design decisions, is a challenging task in practice. Literature provides solutions only for special cases, not including design decisions. Since we have a binding between both artifacts and with the help of a set of predefined constraints on the binding, we can easily validate whether the design conforms to made decisions. If inconsistencies are found, we propose several ways to restore consistency.

Several existing decision management systems define relations between design decisions. These relations could be exploited for guiding developers through the decision space by proposing potentially relevant subsequent decisions and hiding others that are incompatible with already made decisions. However, this is matter of future work.

Chapter 7

Tool Support

The solutions presented in the previous two chapters are conceptual. This chapter outlines a prototypic tool implementation of the concepts to show their technical feasibility. The tool consists of two parts, the implementation of model-independent differences, called *MPatch*, and design decision support for model-based software development. We present the GUI (Graphical User Interface) and architecture of both parts including information about their API, that is, how they can be used and extended. Afterwards, we discuss a set of test cases and their results to show the technical feasibility. Moreover, the tool is used for the validation in Chapter 8.

The project website contains download information, technical details, and tutorials for MPatch as well as the design decision support for model-based software development: `http://modeldiff.imm.dtu.dk`. It also contains installation instructions for the DTU Decision Server, a headless decision management system.

This chapter is organized as follows. Sect. 7.1 gives an overview of the GUI. Sect. 7.2 lists requirements and early design decisions concerning the design of the prototype. Sect. 7.3 outlines the architectural design. Sect. 7.4 and 7.5 present technical details about and APIs of the two parts of the prototype, the implementation of model-independent differences and the decision support in model-based software development. Sect. 7.6 discusses the tests on the prototype along with their results. Sect. 7.7 concludes this chapter.

7.1 The Graphical User Interface

This section covers important parts of the GUI of both parts, MPatch and the design decision support. This is not supposed to be a user guide – tutorials are available integrated in the tool itself as well as on the website. It should rather give the reader an impression of the tool and its functionality without the necessity to install it.

First, the dialogs for creating and applying mpatches[1] are shown. Then, the design decision view is shown, integrated into the modeling environment of an Eclipse-based modeling tool, including a visualization of inconsistencies between a design model and made design decisions.

7.1.1 The GUI for Mpatch Creation

The creation of mpatches is currently implemented with a model-to-model transformation from model-dependent differences of EMF Compare to mpatches. The crucial part is, as stated in Sect. 5.7, the selection and configuration of symbolic references and additional transformations. Figure 7.1 shows the configuration dialog with all available modules loaded, triggered from the EMF Compare GUI that shows the differences between two versions of a model. Finishing this dialog starts the transformations explained in Sect. 5.6 and stores the created mpatch in a file.

The configuration includes the selection of a symbolic reference creator which specifies the matching strategy that will be used when applying the mpatch to another model; id- and condition-based symbolic references are available as explained in Sect. 5.4. The model descriptor creator must also be selected; only one is available at the moment, as explained in Sect. 5.5. The upper part of the dialog shows a selection and the order of additional transformations that are executed after the mpatch has been created. The center part shows detailed information about the currently selected transformation so that developers can see and understand what the respective transformations do.

[1] The term *MPatch* (upper case) refers to the implementation of model-independent differences whereas the term *mpatch* (lower case) refers to an artifact containing model changes.

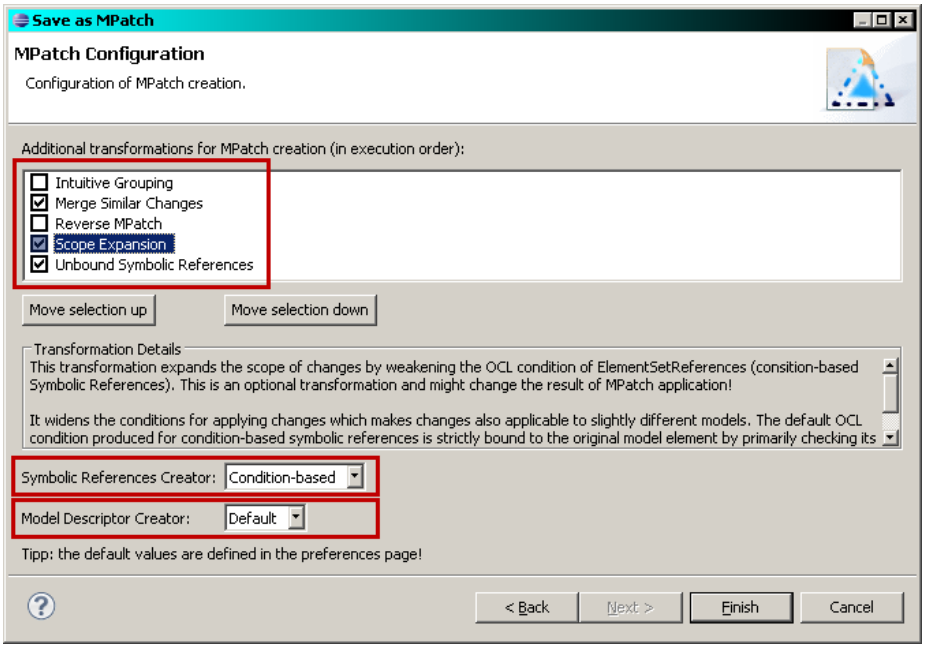


Figure 7.1: The dialog for configuring the creation of an mpatch from model-dependent differences, triggered from the GUI of EMF Compare.

7.1.2 The GUI for Mpatch Application

After an mpatch was created from the comparison of a model, it can be applied to another model. The interesting and most complex part during the application of mpatches is the matching and resolution of symbolic references. The tool supports a user-interactive as well as an automatic way of resolving and refining invalid symbolic reference mappings. Figure 7.2 shows the dialog for the user-interactive resolution of symbolic references.

The tree in the center of the dialog is a representation of an mpatch containing all changes from Tab. 5.12 on page 80 that shall be applied to model M_B, shown in Fig. 5.2 on page 37. Via checkboxes, changes can be excluded from the application; if other changes depend on a deselected change, the other changes will also be deselected (cf. dependency graph in Sect. 5.3.3). The center column shows the number of mapped model elements and the validation result (cf. validation in Sect. 5.8.2). The right-most column lists the set of matched model elements; developers may change that set by clicking on the respective cell.

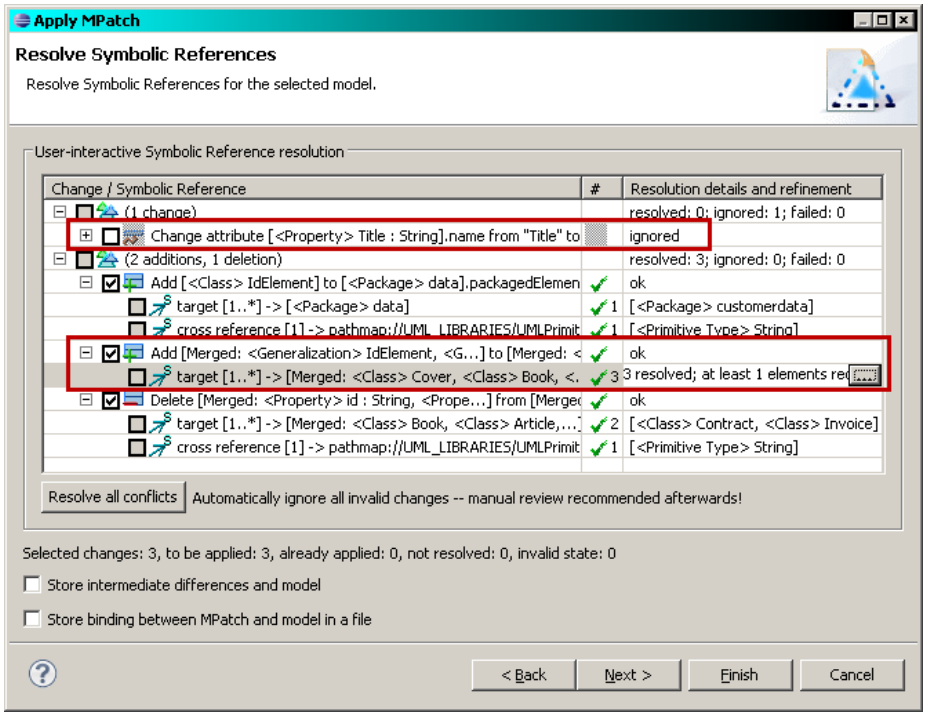


Figure 7.2: The dialog for applying mpatches to a model includes a matching and refinement of symbolic reference mappings.

The first change cannot be mapped to any model element because the attribute *Title : String* does not exist in model M_B. This is why it is not selected for application. All other changes can be applied. However, not all of them are mapped to model elements as intended by the refactoring. The third change describing the addition of a generalization to *IdElement* is currently set to be applied to three model elements, but they are intended to be applied only to *Invoice* and *Contract* (cf. Sect. 5.8.5 on page 87). Clicking on the cell allows developers to refine the set of mapped model elements, for instance, to remove the class *Customer* from this set. The fourth change is matched to the attributes *id : String* of two classes, as intended. So the effort for applying these changes to model M_B is to deselect the first change and to remove one model element from the initial matching.

The button *Resolve all conflicts* triggers the algorithm to automatically resolve all invalid symbolic reference mappings as explained in Sect. 5.8.3. It may also happen that all changes are deselected, if none of them is applicable. The

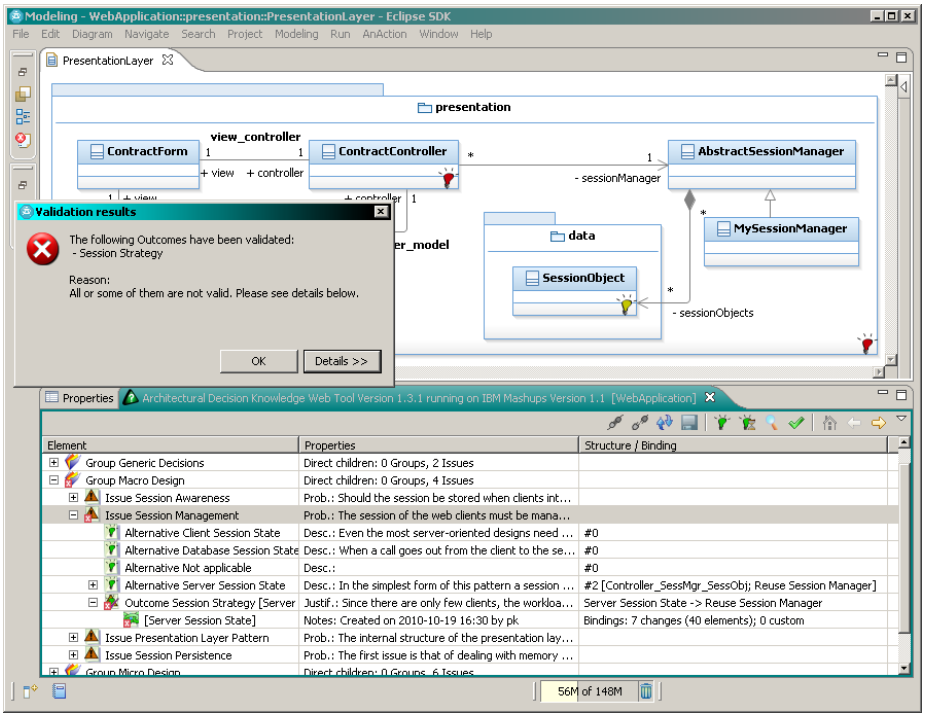


Figure 7.3: The design decision view connected to the ADK Web Tool shows the decisions from the example; some of their bindings are violated.

resulting mapping will be updated in the table and developers may review and refine it.

Moreover, instead of modifying the existing model, the mpatch can be applied to a copy of the model (*Store intermediate differences and model*). Also, a binding between the mpatch and the changed model can be stored in a separate file. More details about the implementation and tutorials are available on the project's website.

7.1.3 The Design Decision View

All tool extensions for supporting design decisions are available in the design decision view. The view is shown at the bottom of Fig. 7.3 and lists all design decisions that are available on the server to which the tool is connected. The

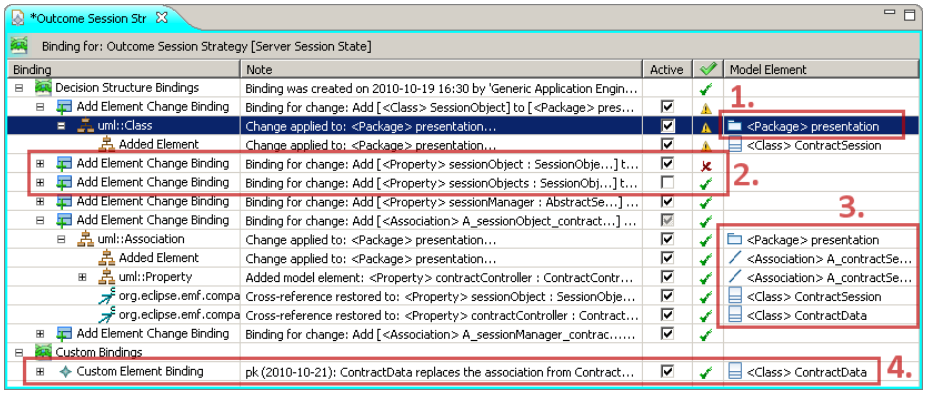


Figure 7.4: A view on the binding shows a detailed list of all causes of the violations and provides opportunities to fix them.

toolbar provides actions for connecting to a decision management system, to make, realize, search for, and validate design decisions. The table below the toolbar visualizes the design decisions as defined in the metamodel in Fig. 6.3, page 103. The decisions are taken from the example in Sect. 2.

Some bindings in this example are violated. These are the same violations as in Sect. 6.6.3 on page 131: an association between *ContractController* and *ContractSession* has been deleted and *ContractSession* has been moved to the package *data*. The violations are visible in several places: the validation dialog provides a compact list; related design model elements are highlighted in the graphical model editor (red and yellow light bulbs); violated bindings are marked directly in the design decision view. The next step is to fix the violated binding.

7.1.4 Fixing a Violated Binding

There are three ways of how a binding can be fixed (cf. Sect. 6.6.2 on page 128): the invalid bindings may be ignored, adjusted, or the model may be corrected. We follow the example from Sect. 6.6.3 and correct the model, adjust the binding, and also ignore parts of the binding.

Figure 7.4 shows a detailed view of the violated binding. All changes that are applied to the model are listed in the view together with corresponding binding notes, validation results, and bound model elements. The following parts are violated and have partially been fixed so far.

1. The package *presentation* is not the current parent anymore for the class *ContractSession* (warnings).
   Fix: This can be fixed by changing the bound package to *data*.
2. Two UML properties cannot be found anymore (errors) – they were part of the associations that has been deleted.
   Fix: This can be fixed by ignoring both bindings (one is already ignored).
3. The deleted association is replaced with a class *ContractData* and corresponding associations (shown in Fig. 6.17 on page 130).
   Fix: The association in this binding has been re-assigned.
4. A custom element binding has been added to also bind the newly added class *ContractData*.
   Fix: This custom constraint was added manually to the binding.

After these four adjustments are done, the binding is updated to suit the model, a subsequent validation succeeds, and the design model is again consistent with all made design decisions.

7.2 Requirements for the Tool Design

This section outlines the requirements for the prototypic implementation. The goal is an academic prototype that shows the technical feasibility of the proposed concepts. We discuss the requirements in the form of early design decisions we made during the development of the prototype.

The first decision concerns the base platform and modeling language, which must conform to MOF. Eclipse[2] is a development platform used by many modeling tools such as the IBM Rational Software Architect and Borland Together but also by open source modeling tools like MoDisco[3] and Papyrus[4]; all of them are based on the same metamodeling language *Ecore*[5] which is a partial implementation of MOF. EMF also comes with a model differencing technology called EMF Compare including a generic and extendable difference calculation algorithm. We decided to use that platform and EMF because of free availability, an active community behind it, and also because of personal expertise knowledge with these technologies. Another requirement was to make the prototype available for others, and the EPL (Eclipse Public License [Ecl04]) is an adequate license for that.

[2]Available at: http://www.eclipse.org
[3]Available at: http://www.eclipse.org/MoDisco/
[4]Available at: http://www.eclipse.org/modeling/mdt/?project=papyrus
[5]Part of the Eclipse Modeling Framework: http://www.eclipse.org/emf/

The second decision concerns the decision management system for maintaining design decisions. Instead of reimplementing the functionality for managing design decisions, the plan is to use existing solutions (cf. Sect. 3.1). The one closest to our decision metamodel (Fig. 6.3 on page 103) is the ADK Web Tool[6]. We decided to specify an interface between our design decision prototype and decision management systems in order to be tool independent and to support more than one decision management systems – this idea was already sketched in Fig. 6.6 on page 107. In fact, our prototype works with two decision management systems at the time being, the ADK Web Tool and the DTU Decision Server.

7.3 Tool Architecture

This section shows the architectural design of our tool and lists all neighboring and our own components. Our tool comprises five components for handling design decisions and the binding between them and design models, and one component (*MPatch*) for specifying and using realizations. Figure 7.5 gives an overview of the architecture. A list and description of all components is given below.

EMF (the *Eclipse Modeling Framework*) provides the metamodeling language *Ecore*; thus, all EMF-based modeling languages are supported, including UML[7].

EMF Compare[8] is a technology for comparing and merging models, used by *MPatch* for the calculation and visualization of model differences.

OCL is used for the specification and its query engine for the evaluation of constraints for consistency checking; the *EMF Validation* framework[9] is used as an adapter to validate these constraints against design models.

Webtools[10] is used for the communication with a decision management system via a web service; the interface is defined later in Sect. 7.5. It provides basic functionality to create, access, and maintain design decisions.

MPatch is used for the creation, representation, and application of model differences as realizations of design decisions.

Decision Manager defines a web service for the communication with a decision management system and maintains the connection. It is also re-

[6] Available at: http://www.alphaworks.ibm.com/tech/adkwik
[7] Provided by UML2 project: http://www.eclipse.org/modeling/mdt/?project=uml2
[8] Available at: http://www.eclipse.org/modeling/emf/?project=compare
[9] Available at: http://www.eclipse.org/modeling/emf/?project=validation
[10] Available at: http://www.eclipse.org/webtools/

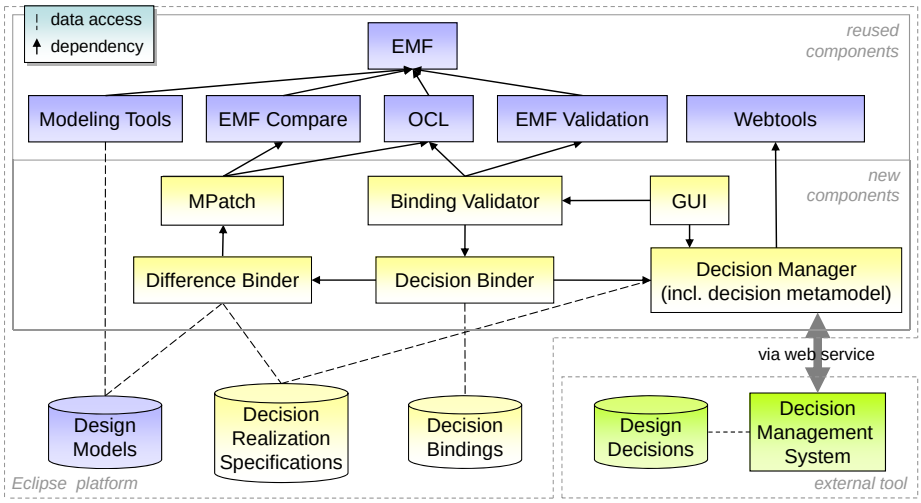


Figure 7.5: The architecture of our tool based on the Eclipse platform; the blue parts are reused components of the Eclipse framework, the yellow parts are our extensions, and the green parts belong to a decision management system.

sponsible for wrapping access to design decisions which are technically located and maintained in the decision management system, and it stores realization specifications of design decisions along with their solutions.

Difference Binder defines the binding between realization specifications of design decisions and the design models to which the changes were applied.

Decision Binder connects bindings to the outcomes of design decisions.

Binding Validator contains logic and constraints for consistency checks.

GUI contains the design decision view and provides dialogs and other user-interactive actions.

Decision Management System stores and maintains all design decisions.

This design is a refinement of the tool setup that was sketched in Fig. 6.6 on page 107. We decided to keep the *Difference Binder* and the *Decision Binder* separate because the former is independent of any design decision – difference bindings can be stored after applying model differences to a model and could also be exploited for activities not related to decision making, for instance, model synchronization.

The API of MPatch and the interface for decision management systems are presented in the two following sections.

7.4 MPatch API

The MPatch API allows tool developers to programmatically use and extend the functionality of MPatch. The provided API comprises operations for all steps listed in the process of creating and applying model-independent differences (Sect. 5.2). MPatch has also five extension points for extending its functionality, for instance, to optimize it for a particular type of models. These extensions are listed in Fig. 7.6. The MPatch *ExtensionManager* maintains them. The prefix *getSelected* means that the extension manager returns the extension which is specified in the MPatch configuration settings.

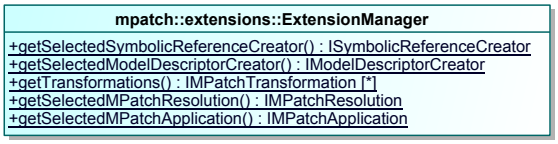


Figure 7.6: The MPatch *Extension Manager* loads and maintains all extensions.

7.4.1 Accessing MPatch Functionality

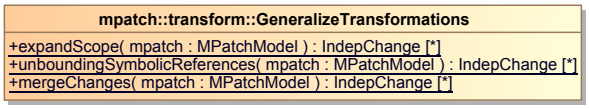
All functions of MPatch can be called programmatically, for instance, to integrate it with other model operations. The second part of our tool, the design decision integration in modeling environments, is an example for using this provided functionality. The remainder of this section explains each of them in detail along with a reference to the section describing the respective concepts.

MPatch Creation (Sect. 5.6)

The *TransformationLauncher* creates an mpatch, is located in the *emfdiff2mpatch* package and requires an *emfdiff* as parameter[11]. The compared models are not required as explicit parameters because they are available via the emfdiff (cf. EMF Compare metamodel in Fig. 5.18 on page 68). A symbolic reference creator and a model descriptor creator are required to perform the transformation and can be retrieved from the extension manager (cf. Fig. 7.6).

[11] EMF Compare also provides an API for comparing two models; for convenience, we wrapped it in an additional operation:
createEmfdiff(leftModel : EObject, rightModel : EObject) : ComparisonSnapshot

MPatch Transformations (Sect. 5.7)



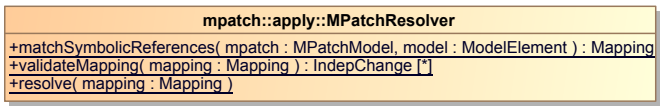
(Sect. 5.7.1)
(Sect. 5.7.2)
(Sect. 5.7.3)

(Sect. 5.7.4)

(Sect. 5.7.5)

Each of the five provided transformations modifies an mpatch and returns references to the modified model changes. They are also registered at the extension manager and can be retrieved via *getTransformations()*. They are explained in the corresponding sections listed next to the transformation signatures.

MPatch Resolver (Sect. 5.8)



(Sect. 5.8.1)
(Sect. 5.8.2)
(Sect. 5.8.3)

The *MPatchResolver* provides operations for matching all symbolic references in a model (*matchSymbolicReferences*), and to validate mappings (*validateMapping*). The mapping is the one defined in Fig. 5.22 on page 83. Then one can either manipulate the mapping to refine or resolve invalid mappings, or *resolve* can be used to automatically resolve all conflicts. However, the automatic resolution ignores all invalid mappings, in the worst case it ignores all changes.

MPatch Applier (Sect. 5.8.4)

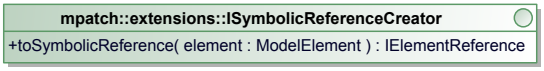
Lastly, if the mapping is valid and not empty, *applyMPatch* applies all changes to the target model. If the binding is created (*storeBinding* is set to *true*), it can be retrieved via *MPatchApplicationResult*. A validation of the binding (required for the validation of design decisions, Sect. 6.5) is possible via the EMF Validation component, because all constraints are registered in that framework:

Diagnostician.INSTANCE.validate(EObject eObject): Diagnostic

7.4.2 Extending MPatch Functionality

One of the requirements for model-independent differences is extendability. The extension manager in Fig. 7.6 already gave a list of all extension points in the framework. Here we briefly explain the role and responsibilities of each of them.

Symbolic Reference Creators (Sect. 5.6.4)



A symbolic reference creator is used during the creation of model-independent differences to replace direct references to model elements with symbolic references in order to make the resulting model changes applicable to other models. The parameter *element* specifies the model element for which a symbolic reference will be created. Exemplary algorithms have already been discussed in Sect. 5.6.4. When a new matching strategy in form of a new type of symbolic references is added to the framework, an appropriate symbolic reference creator must be provided that implements the interface *ISymbolicReferenceCreator*.

Model Descriptor Creators (Sect. 5.6.4)

A model descriptor creator is used during the creation of model-independent differences to replace direct references to added and removed elements in the model with model descriptors in order to make the resulting model changes independent of the compared models. The parameter *element* specifies the model element that will be described. The default algorithm has already been discussed in Sect. 5.6.4. When a new type of model descriptor is added to the framework, an appropriate model descriptor creator must be provided that implements the interface *IModelDescriptorCreator*.

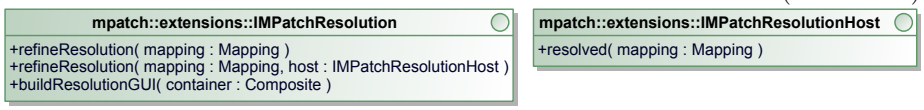
Transformations (Sect. 5.7)

A transformation on an mpatch may perform any in-place modification. We have seen examples for generalizing model changes, grouping or reversing them. A transformation must return the set of changes which have been modified.

MPatch Application (Sect. 5.8.4)

This extension provides the algorithm for applying model-independent differences to a model. The framework checks beforehand that the *mapping* is valid (cf. *MPatch Resolver* in Sect. 7.4.1). When a new change type is added to the framework or model-specific applications of changes are required, an adequate application algorithm must be provided, for instance, by extending the default one.

MPatch Resolution (Sect. 5.8.3)



This extension for resolving invalid mappings during the application of model-independent differences must provide two strategies: an automated, non-interactive strategy (*refineResolution(mapping)*), and a user-interactive strategy (*refineResolution(mapping, host)*). When a new resolution strategy only realizes either of the *refineResolution* operations, then it should delegate the other to the default implementation. Our default implementation of this extension provides already both strategies (cf. GUI explanations in Sect. 7.1.2). The interactive resolution works as follows. First, the framework calls *buildResolutionGUI(..)* so that the GUI for the interactive resolution can be set up. Second, *refineResolution(mapping, host)* is called by the framework and gives control to the resolution implementation. Third, whenever the implementation modified the mapping, it must call *resolved(mapping)* on the *host*. Other strategies are possible as well, for instance, providing a smarter way of automatically resolving invalid mappings.

This section presented the API and extension points of the MPatch technology including the information for tool developers to integrate MPatch in other tools or to extend it. For instance, to optimize it for other types of models.

7.5 Interface to Decision Management Systems

Our tool uses decision management systems for storing and maintaining design decisions. This section specifies and explains an interface which decision management systems must implement before they can be used with our tool. The first purpose of the interface is the management of all design decisions from

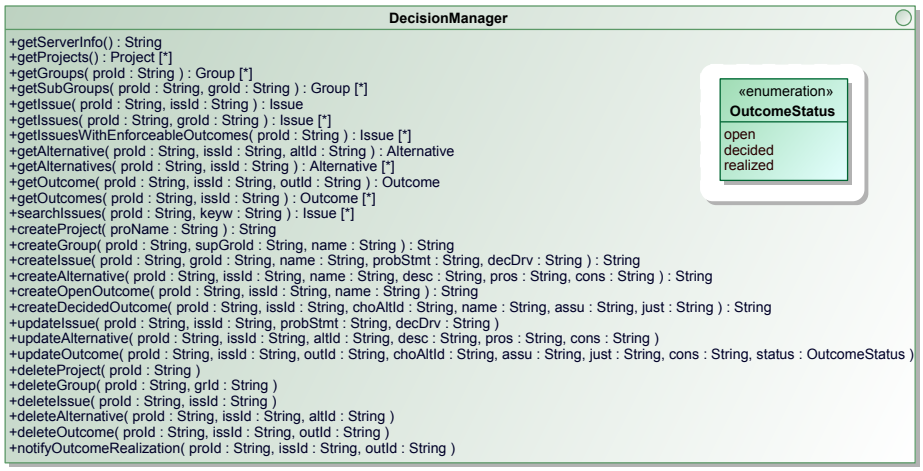


Figure 7.7: The interface for decision management systems.

within the modeling tool; switching back and forth between the two tools can be avoided this way. The second purpose is to maintain the life-cycle of a decision, that is, to update its status properly. The functionality is motivated by the scenarios that were discussed earlier in Sect. 6.2.2, page 105.

The *Decision Manager* interface in Fig. 7.7 makes all decision handling activities specified in Sect. 6.2.2 available in our tool. The UML interface shows all operations of the web service interface and is available as a WSDL file[12]. All getters, updates, creates, and deletes operate on the decision model that was explained earlier in Fig. 6.3 on page 103. The return values of all create-operations are unique identifiers of the respective created entities.

Making a design decision (activity 2a in the decision handling scenario, Fig. 6.5 on page 106) for the example in Sect. 2, for instance, could be performed by calling *createDecidedOutcome*: *proId* and *issId* identify the current project and issue, respectively, while *choAltId* specifies which alternative was chosen. The remaining parameters specify a *name* for the outcome, *assu*mptions, and a *just*ification. After the outcome has been realized in the design model, *notifyOutcomeRealization* updates the status of the decision from *decided* to *realized*.

Except for *updateOutcome* and *notifyOutcomeRealization*, all operations are stateless, that is, their semantics is only scoped to single operation calls. The

[12]The WSDL (*Web Service Description Language*) file contains a specification of the interface and is available online at: `http://modeldiff.imm.dtu.dk/wsdl`

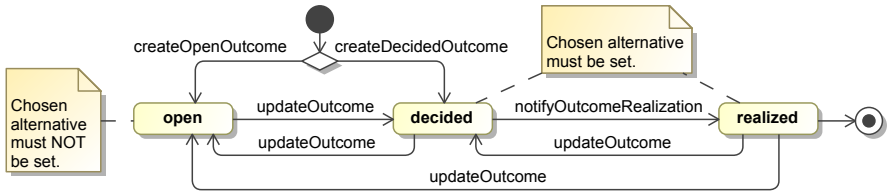


Figure 7.8: Valid status transitions for design decision outcomes at our interface; the status may also be updated by decision management systems.

status of an outcome defines the life-cycle of a decision, as depicted in Fig. 7.8: status transitions must follow the given automaton. *UpdateOutcome* may be used at any time, also to manually correct a status. However, tool implementations should rather update the status by invoking the operations *createOpenOutcome*, *createDecidedOutcome*, and *notifyOutcomeRealization*; an example is the sequence *createDecidedOutcome* and *notifyOutcomeRealization* as discussed in the previous paragraph.

Different decision management systems may, of course, have their internal strategy of handling the life-cycle of an outcome. In these cases, their realization of this interface must include a mapping to the internal strategy. The ADK Web Tool, for instance, has a one-to-one mapping of the similarly named states except for its additional state *rejected*, which does not exist in our interface. That status, however, is mapped to *open*, because a rejected decision in this particular decision management system can be *decided* again, just like open outcomes.

For the time being, we know two decision management systems which implement this interface: the ADK Web Tool [ZGK+07] as well as the DTU Decision Server, a headless decision management server. This shows that the WSDL interface enables interoperability and multi platform support without much development effort.

7.6 Testing the Prototype

The GUI and the architectural design of the tool do not yet give any evidence whether the concepts work. This section presents a set of test cases that prove for a number of test models that the implementation of model-independent differences works as expected. Another set of test cases validates the correct behavior of the decision management system which implements the interface specified in Sect. 7.5.

Test case	Tests	✔/✘	UML	EMF	Eachonce[13]	Complete
Transformation emfdiff to mpatch	5	5/0	✘	✘	✔	✘
Correctness of OCL expressions	5	5/0	✔	✔	✔	✘
Matching	3	3/0	✘	✘	✔	✘
Individual change types	9	9/0	✘	✘	✔	✔
Dependencies	2	2/0	✘	✘	✔	✘
Generalizations (only cond.-based sym. ref.)	11	11/0	✘	✔	✔	✘
Grouping	3	3/0	✘	✘	✔	✘
Reversal	7	7/0	✔	✔	✔	✔
Detection of already applied changes	14	14/0	✔	✔	✔	✔
UML specifics and complex test	3	3/0	✔	✘	✘	✔
EMF specifics and complex test	2	2/0	✘	✔	✘	✔
Eachonce specifics	2	2/0	✘	✘	✔	✔
Performance test	1	1/0	✘	✔	✘	✔

Table 7.1: Test cases and results for MPatch; *Complete* denotes a complete run through the testing process in Fig. 7.9.

7.6.1 MPatch Tests

The development of model-independent differences was tested with 67 regression tests covering all non user-interactive features. Table 7.1 gives an overview of all test cases, their results, and the types of models which they cover. The column *Complete* denotes a test specification that involves all steps in our testing process which is shown in Fig. 7.9 and explained below.

More than half of the tests run through a sophisticated testing process that is shown in Fig. 7.9. It consists of six steps and after each step, several assertions test whether the step was finished successfully. The input of this process are an unchanged version M_A and a changed version M'_A of a model from which an mpatch will be created. Optionally, another model M_{A2} may be specified to which the mpatch will be applied and the resulting model M''_{A2} will be compared against another M'_{A2} (otherwise, M_A and M'_A are used). Moreover, the configuration must be specified, that is, the type of symbolic references (default: all types are tested), and a set of transformations θ_2 that shall be executed. The following list explains each step in detail. Trivial assertions are ignored below, for example, whether a return value is *null*.

[13] *Eachonce* is a modeling language designed for the tests that covers all features that the meta modeling language provides, including multi-valued attributes.

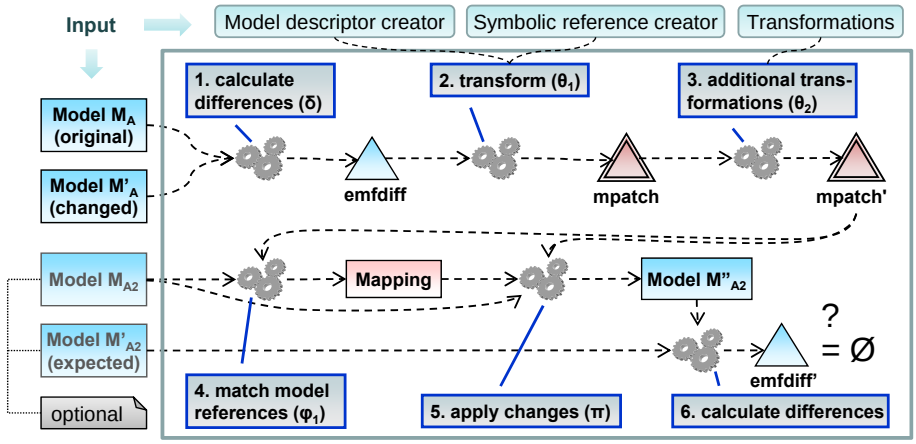


Figure 7.9: The complete testing process comprises six steps and several assertions after each step; if M_{A2} and M'_{A2} are not set, M_A and M'_A are used instead.

1. EMF Compare calculates all differences between the two input models M_A and M'_A and returns an *emfdiff*.
   Assertion: the *emfdiff* must contain at least one change.
2. Transform *emfdiff* to *mpatch* with the specified symbolic reference creator.
   Assertion: *mpatch* must be valid (tested with EMF Validation).
   Assertion: *mpatch* must contain the same number of changes as *emfdiff*.
3. Execute transformations θ_2 as specified; this always includes the creation of the *dependency graph* and *grouping*.
   Assertion: at least one group must be created.
   Concrete test cases may add further assertions for other transformations.
4. Match *mpatch'* in model M_{A2} (or M_A).
   Assertion: the mapping must be valid (all changes must be applicable).
5. Apply *mpatch'* according to the initial mapping (also create binding).
   Assertion: the application result must not contain incomplete or failed changes.
   Assertion: the *binding* must be valid (tested with EMF Validation).
6. EMF Compare calculates all differences between the actual result M''_{A2} and the expected result M'_{A2} (or M'_A).
   Assertion: the actual and expected models must not differ, thus, the computed *emfdiff'* must be empty.

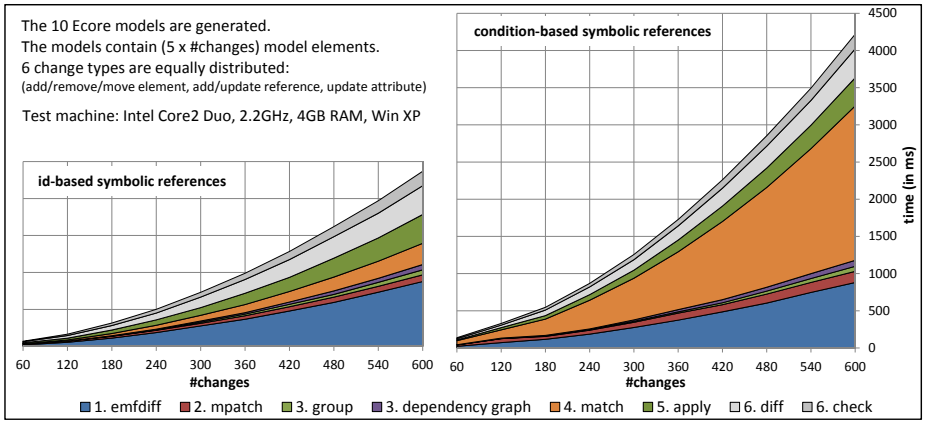


Figure 7.10: Results of the performance test on generated models comprising between 60 and 600 model changes; the numbers are average values of ten test runs.

This automated testing process is used for most test cases and covers all functionality of model-independent differences except for the user-interactive parts. There are fine-grained test cases for each change type (*individual change types* in Tab. 7.1) and functionality so that failing tests are clear indicators of which part is broken. The *complex tests* include up to 18 changes that also interfere with each other. As a side effect, the tests also ensure that the third-party tool EMF Compare works as expected.

The performance test automatically creates a set of Ecore models, that is the modeling language of EMF, with increasing size and increasing number of model changes. Then, the entire testing process is run for each of these models. This has been done for both, id-based and condition-based symbolic references, the results are shown in Fig. 7.10. The time is separated for the individual steps of the testing process; the total time for 600 changes with condition-based symbolic references, for instance, takes slightly more than four seconds whereof the largest part is the matching step with approximately two seconds. The condition-based and id-based performance results mainly differ in the times for symbolic reference matching.

We can learn several aspects from these results. First of all, the creation of an mpatch (steps 2–3) is very fast, much faster than the creation of an emfdiff (step 1). Further, the matching strategy has a major influence on the performance. Id-based matching is extremely fast; condition-based matching, on the other hand, uses an OCL engine to perform the matching. Parsing the OCL

expressions, creating the query, and executing it on the model are the most time-consuming operations.

Overall, the measured times are acceptable in practice. We do not expect design decision realizations to consists of more than 100 changes, from our experience most realizations were around 5–20 changes. Another important factor is the model size – the bigger the model, the longer the matching. We tested the matching of 10 changes with condition-based symbolic references on a UML model with more than 5.000 elements and it took only 2–3 seconds, which is again fast enough in practice. However, the complexity can be reduced by performing the matching not on the entire model but on a submodel, for instance, on only one of the packages in a UML model.

7.6.2 Decision Interface Tests

The interface introduced in Sect. 7.5 allows the use of different decision management systems. In order to validate that a decision management system works as expected, we defined a test suite containing automated test cases that test the communication. They also validate all the functionality that our tool requires from a decision management system.

Test case	**Tests**[14]	✓/✗ ADK W. T.	✓/✗ DTU D. S.
Connectivity and server info	1	1/0	1/0
Manage projects	1 (2)	**0/1**	1/0
Retrieve decisions	2 (10)	2/0	2/0
Create decisions	1 (5)	1/0	1/0
Update decisions	1 (4)	**0/1**	1/0
Delete decisions	1 (5)	**0/1**	1/0
Outcome status updates	1 (4)	1/0	1/0
Search for decisions	4	4/0	4/0

Table 7.2: Test cases for the decision interface and results for the ADK Web Tool and the DTU Decision Server.

Table 7.2 gives an overview of all test cases; they systematically test the operations of the interface in Fig. 7.7 on page 152. With one exception, all test cases operate on a dedicated test project on the decision management system in order to not interfere with existing projects. The exception is the test case *Retrieve decisions*, because it retrieves all decisions from all projects on the server to cope with more data than just the data specified in the test cases. All test cases are self-checking; for instance, after creating a new decision, the created decision is

[14]The values in parentheses denote the number of different interface operations involved in the respective test case.

again retrieved to verify its creation. The test case *Outcome status updates* is important because it verifies that the decision management system's behavior matches the behavior we specified in Fig. 7.8 on page 153.

Both tools, the ADK Web Tool as well as the DTU Decision Server, support all required operations for capturing, making, and realizing design decisions. However, the ADK Web Tool does not support maintenance operations for design decisions. The reason is that maintenance should rather be performed inside the ADK Web Tool, not in the modeling tool.

7.7 Conclusion and Discussion

The purpose of this chapter is to show the technical feasibility of the conceptual contributions from Chapters 5 and 6 by presenting a tool that implements these concepts. *MPatch* is the name under which the implementation of model-independent differences are contributed to the open source project EMF Compare which is part of the Eclipse project. The decision support tool is published on the project websites. Both tools are freely available as open source projects.

We presented parts of the graphical user interface showing important dialogs of both implementation parts. Then we explained the tool architecture and some of the rationale behind it. Since one of the requirements is extendability, we summarized the API and extension points of model-independent differences and explained the interface for decision management systems. The latter allows using different decision management systems with our tool.

In order to provide evidence that the implementation works as expected, we defined a sophisticated testing process for our tool. A set of in total 67 test cases covers non user-interactive functionality of MPatch, and 12 test cases validate whether decision management systems implement the interface as expected.

To conclude, the implementation serves as a proof of concepts. MPatch became a generic tool for patching models and transferring arbitrary changes between models.

Chapter 8

Validation

The tool presented in Chapter 7 implements the concepts from Chapters 5 and 6 and also includes a set of test cases to ensure that the functionality works as expected for all test models (cf. Sect. 7.6). Although these tests are a proof of the technical feasibility, they do not give any evidence whether the approach is useful and applicable to other than the presented examples and in real projects with non-artificial models. Because of this, we tested our approach with more realistic examples, design patterns and refactorings as design decisions, and we performed domain level tests in which we applied our approach to a real project. This way, we show that the goals from Chapter 4 are met to a large extent. In addition, we interviewed six architects and modelers and got valuable feedback, especially concerning practical usability and success factors.

A proper evaluation would include an analysis of documenting, capturing, making, and validating design decisions during the development of a running project. Only then, it would show whether the additional effort for capturing decisions and their realizations pays off and automated realization and validation really support developers. However, there was not enough time for applying our concepts in such a running project. Therefore, this section is only a *validation* of our concepts by demonstrating the applicability to realistic examples and in a replay of a real project. An evaluation in a running project is future work.

8.1 Applicability

Design patterns are frequently used fragments in the design of software systems, and could, consequently, be the result of design decisions [HAZ07]. We used model-independent differences to capture realizations for all 23 design patterns from the Gang of Four [GHJV95], ran the generalization transformations, and evaluated their applicability to at least two other models. Refactorings are modifications of design artifacts to improve their overall structure without changing the behavior or functionality of the system. We also captured 25[1] refactorings [Fow99] as solutions for design decisions, and evaluated their applicability to at least two other models.

Although there are specialized approaches for applying design patterns and refactorings, they typically require a big specification effort, for instance, in a special modeling language or by programming the respective modifications [RSA10, KPRP07]. Our approach should rather be used to extract such modifications as easy and fast as possible from an example and, at the same time, being able to also apply it to other models.

The results of the two following subsections are that already half of the examples (design patterns and refactorings) could be captured as reusable design decisions including their realization specifications without additional manual effort. The other half requires manual adjustments either of model differences or of the model after realizations have been applied; however, the manual adjustments are still less effort than modeling the realization manually. Only a small number of cases could not be expressed in our tool. This validates goal 2: automation of recurring design decision realizations.

8.1.1 Design Patterns as Design Decisions

A design pattern shall have a meaningful name, explains a solution for a specific problem, and discusses the context and consequences [GHJV95]. We used the generic examples (which are part of the described solutions of design pattern descriptions) as a basis for example models from which we captured realization specifications for design decisions. Then we tried to apply each of the captured patterns to at least two models which comprise the problem that the pattern addresses. For example, one of the models for applying the *Observer* design pattern contains a class *TextField* and a class *Label* – the latter is the observer and reacts on changes of the text field, which is the subject of observation. The

[1] The other refactorings are only applicable code, not to models.

Design pattern name	#changes	Status
1. Abstract Factory	9	*E*
2. Builder	5	*E*
3. Factory Method	2	*E*
4. Prototype	4	*G*
5. Singleton	5	*G*
6. Adapter	2	*E*
7. Bridge	5	*E*
8. Composite	5	*G*
9. Decorator	4	*E*
10. Façade	4	*E*
11. Flyweight	12	*E*
12. Proxy	2	*E*
13. Chain of Responsibility	4	*G*
14. Command	5	*E*
15. Interpreter	5	*G*
16. Iterator	7	*G*
17. Mediator	6	*L*
18. Memento	6	*G*
19. Observer	6	*G*
20. State	5	*E*
21. Strategy	5	*E*
22. Template Method	3	*E*
23. Visitor	7	*E*

Status	Cases	Meaning
G	8	Pattern is (G)enerically applicable.
E	14	Pattern is applicable but produces (E)xample pattern in the model.
L	1	Pattern is not generically applicable; (L)imitation.

Table 8.1: Using design patterns as design decisions.

design decision realization was captured from an example model with differently named classes though.

The results of this experiment are listed in Tab. 8.1. Eight of the 23 design patterns could be captured and applied to other models right away without adjusting anything; they are marked with a *G* in the table.

14 design patterns could be captured and applied to other models, but the resulting models did only reflect the concrete examples from which the patterns were captured. Hence, the resulting models had to be adjusted to fit the context; these cases are marked with an *E* in the table.

One case, the mediator pattern, could not be captured properly, marked with an *L* in the table. The reason is that our tool is currently not capable of adding an association multiple times to the same model element but with different targets. At the moment, a change can only be applied multiple times to *different* model elements.

Refactoring name	#changes	Status
1. Move Method	1	*G*
2. Move Field	1	*G*
3. Extract Class	5	*C*
4. Inline Class	5	*C*
5. Introduce Local Extension	2	*G*
6. Replace Data Value with Object	4	*G*
7. Change Value to Reference	3	*G*
8. Change Reference to Value	3	*G*
9. Duplicate Observed Data	8	*C*
10. Change Unidirectional Association to Bidirectional	2	*G*
11. Change Bidirectional Association to Unidirectional	2	*G*
12. Replace Type Code with Class	8	*L*
13. Replace Type Code with Subclass	5	*L*
14. Replace Type Code with State/Strategy	10	*L*
15. Add Parameter	1	*G*
16. Remove Parameter	1	*G*
17. Introduce Parameter Object	4	*G*
18. Hide Method	1	*G*
19. Pull up Field	2	*L*
19. Pull up Method	2	*L*
21. Push down Field	1	*G*
22. Push down Method	1	*G*
23. Extract Subclass	2	*C*
24. Extract Superclass	4	*C*
25. Extract Interface	4	*C*

Status	Cases	Meaning
G	14	Refactoring is (G)enerically applicable.
C	6	Movement was not (C)aptured by EMF Compare.
L	5	String transfer between changes does not work; (L)imitation.

Table 8.2: Using refactorings as design decisions.

To conclude, a third of the tested design patterns could be reused without any additional effort. With one exception, all others are at least beneficial in the sense that they support developers by automatically creating example realizations. This is also helpful for unexperienced developers because they already see an example of the solution in their models.

8.1.2 Refactorings as Design Decisions

Similar to design patterns, refactorings address a particular problem but this problem is typically a bad or unclear design. Bad smells and anti patterns [BMMM98] indicate the necessity for refactoring a design; that is, existing design artifacts are restructured to improve the design. The aforementioned literature recommends frequent refactorings during the design process to keep the design comprehensible and extendable. Hence, whenever developers decide to restruc-

ture the design, capturing these refactorings as design decisions documents the task and provide conformance checks without additional effort.

We captured 25 refactorings [Fow99] (those that are applicable to models) as realization specifications for design decisions and applied each of them to at least two other models which comprise the problem that the refactoring addresses. The results of this experiment are listed in Tab. 8.2. 14 of the 25 refactorings could be captured and applied to the other models right away without adjusting anything; they are again marked with a *G* in the table.

Eight of the captured refactorings did not properly represent the changes that were made; for instance, EMF Compare could not detect the movement of existing attributes to a new class in the refactoring *Inline Class*. Instead, it detected deletions and additions of the respective attributes. After adjusting the realization manually, however, the refactoring could properly be applied to other models; such cases are marked with a *C* in the table.

Again, three of the tested refactorings could not be expressed with our tool. An example is the replacement of an attribute with an equally named class, which requires the transfer of a string value of a deleted element (attribute's name) to a newly added element (class's name). Such cases are marked with an *L* in the table.

To conclude, most (22/25) of the refactorings could be expressed with model-independent differences. However, the difference calculation algorithm of EMF Compare was not precise enough to properly detect all changes. Using other techniques like an operation recorder as discussed in Sect. 5.6 could overcome this problem. Although some of the realizations seem to be trivially simple with just one to three changes, they might, nevertheless, result in effective changes in design models, especially because changes are applicable to multiple model elements. However, we also encountered cases in which the expressiveness of model-independent differences was not sufficient. An extension for expressing relations between changes would tackle that issue and is future work.

All goals from Chapter 4 could partialy be validated with these tests: the documentation of design decisions is directly linked to design models, slightly more than half of the design decision realizations could be automated right away, and consistency checks are available without additional effort.

8.2 Domain Level Test

The example in Sect. 2 is artificial and its purpose is the concise and compact illustration of the conceptual solution throughout the thesis. Design patterns and refactorings are also just small examples. The question is whether and how the solution works with bigger models in real projects and with related design decisions? We investigated an experience report dealing with the design of an SOA (*Service Oriented Architecture*) application [ZDGH05] and replayed several design decisions. One of the authors answered questions about the rationale behind the design so we could model eleven significant design decisions based on that information. The decisions may not necessarily be made as presented here; this is a controlled experiment to validate the applicability of our tool to an architectural model with a more realistic size than the aforementioned examples. Hence, all three goals are validated in respect of making, realizing, and validating a sequence of realistic design decisions.

The project deals with an architectural design of a server application for a big telecommunication wholesaler. The application must provide two fundamental business processes to two types of clients, web browsers and other systems via web services. The bigger of the two is the move of an existing customer to a new address and consists of nine complex activities that are available to the clients. These activities are again composed of more than 100 subactivities.

The architectural design of the system comprises seven layers [ZDGH05]. We modeled it as a UML component diagram in Fig. 8.1 in which we placed the second and third lowest layers on the same level. The business process layer contains a business process execution engine including a specification of all nine activities. Subactivities are realized in three further layers below the business process layer because they are partitioned into application-specific (for application services) and application-generic (for business services) layers. The lowest layer represents existing systems for data storage and additional core systems. The upper two layers comprise the communication with clients. We modeled several further diagrams which refine the internal structure of components or entire packages – some of them are depicted and refined below. The realizations of all discussed decisions are stepwise applied to that design model.

We modeled eleven design decisions and the aforementioned author confirmed them to be relevant architectural decisions. In addition, we captured their realizations in the design. All decisions concern the interaction of the business process layer with its neighbor layers, including the selection of a reference architecture which is responsible for executing business processes, and the data structure that is used for the activities. Table 8.3 summarizes all decisions and states their outcomes without justifications. For each of the decisions marked

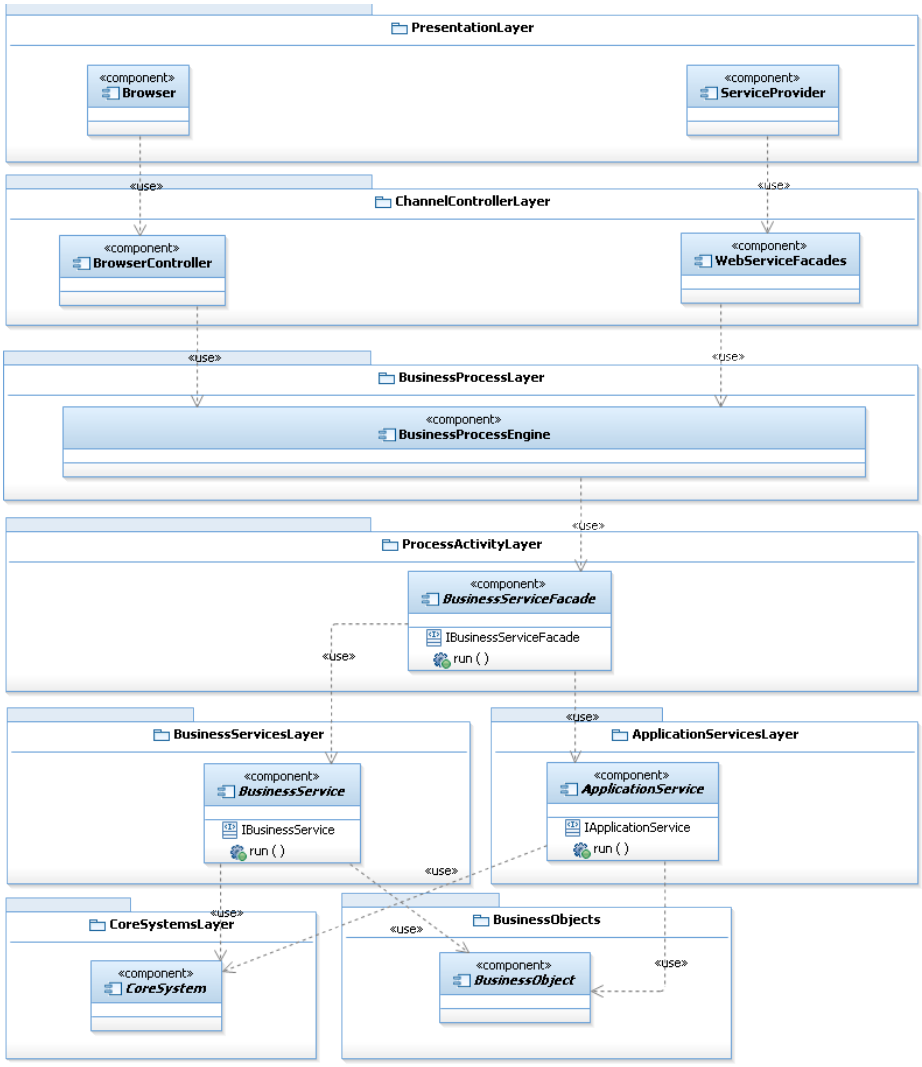


Figure 8.1: The system's architecture shown as a UML component diagram; this is an abstracted view from the report [ZDGH05].

with (*), the realization in the design is presented below. The first decision is explained in detail and all other decisions are only briefly described because we focus on their realization in the design model.

8.2.1 Decision 1: Provided Activities

Problem Statement. Business processes consist of several activities that are typically executed in a specific order. These activities may be automatically executed, they may depend on other internal events, and/or they are triggered by a client. This decision deals with how the latter type of activities, that are provided to the clients, shall be realized in the architectural design.
Decision Drivers: Reusability, extendibility, loose coupling.

Alternative: Common Interface. A common UML interface defines generic operations for all activities to provide their respective functionality to the clients. Each of the activities that offers some functionality to the clients must implement that interface; this results in a uniform way of delegating client requests to the activities of a business process. Example operations for such an interface are:

callSync(data : Array of Object): **boolean**
callASync(data : Array of Object): **int**
getResult(requestId : **int**): Array of Object

Pros/Cons: The advantage is that each activity can be treated the same by the presentation layer, no matter what the activity's functionality is. The downside is that the activity's functionality must be mapped to the interface, which may not always easily be possible.

Another alternative is, for instance, the façade design pattern for provided activities.

Outcome: Common Interface. Since the activities are designed from scratch, it is easily possible to incorporate a specific interface for providing their functionality to clients. Both, synchronous and asynchronous calls are required with arbitrary parameters, so the realization will exactly be the interface explained in the alternative.

#	Brief description of the design decision
1 * (R)	*Provided Activities:* Several activities in the business process layer shall be provided and visible to clients, others shall not. If an activity is visible, there shall be a uniform way of accessing it. *Outcome:* Provided activities must implement a specific interface
2 (R)	*Activity Delegation:* Several activities in the business process layer are realized in lower layers, others are not. The former delegate the operation to a lower level, which shall be realized in a uniform way. *Outcome:* Delegated activities must implement a specific interface
3 * (R)	*Business Process Engine (BPE):* The business process layer requires a business process execution engine to interpret and run business processes. *Outcome:* Reference architecture by Leymann and Roller [LR99]
4 (P)	*Component Structure in Business Process Layer:* The reference architecture comprises 18 components; these components can either be organized as subcomponents of the BPE or as high-level components in the layer. *Outcome:* Subcomponents of the BPE
5 * (N)	*Database for BPE:* The BPE requires a database for storing the business processes and states of their instances. *Outcome:* (since other layers also require databases, this decision is postponed because the same database may be used in several places)
6 * (P)	*Queues in BPE:* Queues are used to buffer messages in a service oriented architecture; sometimes they are not relevant on a particular level of abstraction. *Outcome:* Removal of queues from the current design
7 (N)	*Authentication and Security:* Only authorized persons shall have access to the data, and communication must be secure, for instance, encrypted. *Outcome:* (out of scope in the current design phase)
8 * (R)	*Data Modeling:* Most activities in the business process require data as input and produce data as output. A uniform way of data definition shall be implemented. *Outcome:* Input and output data containers
9 (R)	*Data Maps:* The output data from one activity may be the input of another activity. This mapping must be realized in the design. *Outcome:* Function for input data containers
10 (R)	*Session Management in BPE:* The state of business process instances (application state) must be stored. This includes active activities and the values of data containers. *Outcome:* Database session state in business process layer
11 (R)	*Session Management in Web Channel:* The communication between the browser client and the presentation layer includes several forms. A session could store the data to allow multi-page forms. *Outcome:* Client session state

Table 8.3: An overview of all eleven design decisions we modeled for this system; R – realization specification captured, P – project-specific realization captured, N – no realization captured; * – realization is explained in the text.

Realization. There is no existing realization specification from previous projects or other example models. Because of that, we recorded[2] the model changes for three activities to specify a new realization:

First, we recorded the changes shown in the simplified model in Fig. 8.2: a new interface *IActivityStub* has been added to three existing activities that are provided to clients.

Second, we automatically extracted the design model changes with the help of MPatch. The configuration includes *Scope Expansion*, *Merge Changes*, and *Unbounding References* (cf. Sect. 5.7). The result are four changes as shown in the upper part of Fig. 8.3:
(1) new interface *IProvidedActivity*,
(2) new class *ProvidedActivityStub*,
(3) new interface realization *IProvidedActivity* between these two elements,
(4) a reference change (generalization) from *ActivityStub* to *ProvidedActivity-Stub* applicable to classes called similarly to *ActivityStub*.

Third, we applied these changes to the unchanged design and in the dialog for resolving symbolic references (see also Fig. 7.2 on page 142), we select all activities that should be provided to the clients. In the end, a new realization specification has been added to the chosen alternative, and a binding connects these two artifacts to enable traceability and validation, indicated in the lower part of Fig. 8.3.

Next, we compare the effort for realizing the decision with and without our tool. Without the tool, we had to realize the decision manually in the design and there is no link between the documented decision and the design. With the tool, we also had to realize the decision manually (because there was no suitable realization specification available for the chosen alternative), and in addition we had two additional dialogs: one for specifying a new realization (creating an mpatch) and one for applying it to the design. However, the binding between the outcome of the decision and the affected design model elements can be used to automatically validate the consistency between both artifact with respect to the decision, instead of doing that manually. Moreover, the realization that we captured for the chosen solution is now reusable, for instance, in other projects.

[2]The recording feature is part of our tool and works as follows: when a user starts the recording, a copy of the parts of the model the user is working on is stored in memory; when the user finishes the modifications, another copy is made of the changed version and the process in Sect. 5.2 is started.

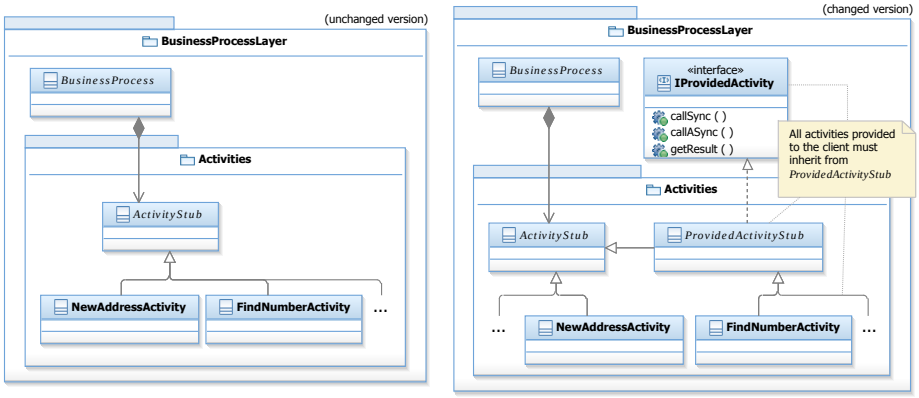


Figure 8.2: An excerpt of the design model before (left) and after (right) the realization of decision 1: Provided Activities.

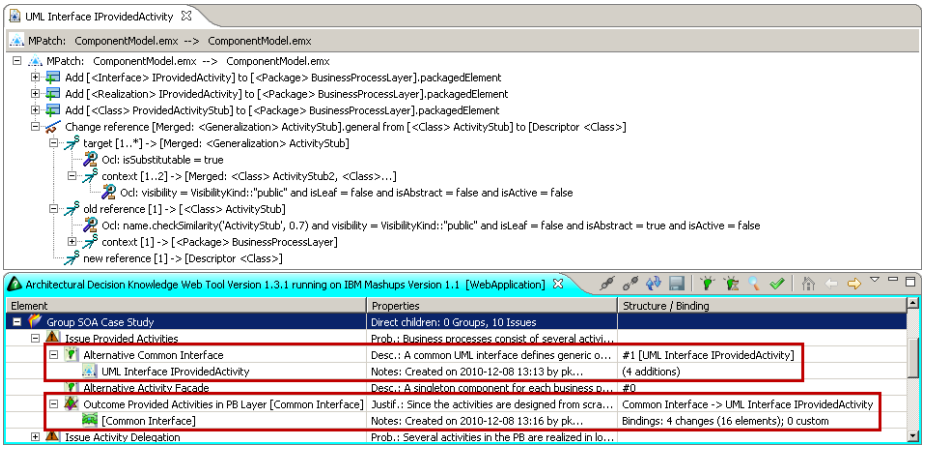


Figure 8.3: The realization specification of decision 1 includes a UML interface *IActivityStub*, a new class *ProvidedActivityStub*, and some updated inheritance relations.

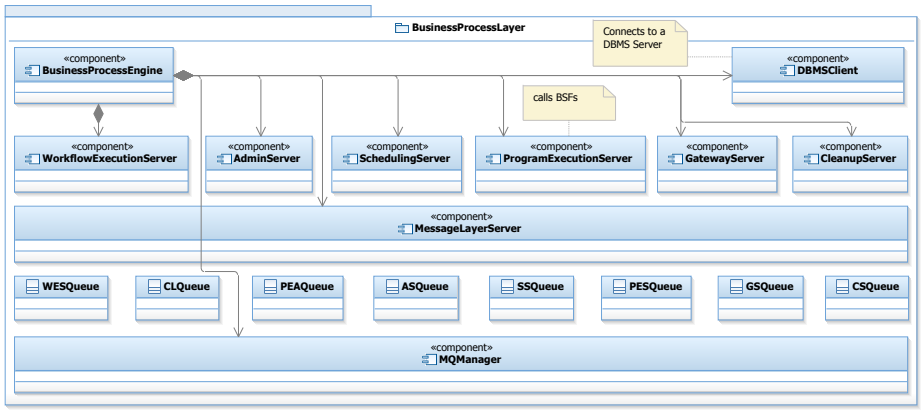


Figure 8.4: The chosen reference architecture for a business process engine applied to the design model (decision 3); the containment relations were added in decision 4.

8.2.2 Decision 2: Activity Delegation

This decision concerns activities that are realized in the layers below the business process layer. The issue is how to model the delegation from activities to business service façades (BSF). We have chosen a similar solution as in decision 1, namely to introduce an abstract class that handles the delegation. Therefore, we skip the detailed description of this decision.

8.2.3 Decision 3: Business Process Engine

This decision concerns the choice of a business process engine (BPE) for executing all business processes in the business process layer. The choice is the workflow reference architecture by Leymann and Roller [LR99] that describes a server-client architecture for business workflow execution. The server component comprises 18 components which we would like to realize in the design.

Realization. For this decision, we prepared a realization in a dedicated example model based on the specification of the BPE. Then we used MPatch to specify a realization for the chosen solution by extracting 18 changes, one for each component of the BPE. This realization specification can now be used every time this particular BPE is chosen.

We applied that realization to the business process layer package of the design model, the result is shown in Fig. 8.4 (excluding the containment relations which are part of decision 4). No violations occurred. The effort was very low because the MPatch technology created all components automatically. We only had to choose the package to which the realization should be applied. And again, the binding between the decision outcome and the newly created design model elements can be used to validate consistency.

8.2.4 Decision 4: Component Structure in Business Process Layer

This decision is a refinement of the previous one. The chosen BPE comprises several components but it does not specify how they are organized. Instead of having high-level components in the package, we decided to add containment relations to the main component *BusinessProcessEngine* as modeled in Fig. 8.4. The rationale behind this decision is a clear design and that the BPE component is responsible for organizing all subcomponents.

Realization. The realization is straightforward and consists of nine changes describing the addition of a composition association from *BusinessProcessEngine* to each subcomponent. No violations occurred. In this special case, we could not create a single generalized model change because MPatch does not allow a single change to be applied multiple times to the same model element (here: adding nine associations to the package *BusinessProcessLayer*). Therefore, this realization is project-specific and not suitable for reuse in other projects. However, the binding between the decision outcome and the associations can still be used for consistency validation.

8.2.5 Decision 5: Database for BPE

A consequent issue after decision 4 is the choice of a database for the BPE to store business processes and their instances. This is required for the component *DBMS Client* (cf. Fig. 8.4). However, we postpone this decision because there are other places in the design which might require a database. It would be cheaper to share one database between several components of the server application instead of setting up several individual databases for each component that needs one.

8.2.6 Decision 6: Queues in BPE

The chosen BPE includes eight queues for the communication with BPE clients (shown in Fig. 8.4). These technical details are not relevant in the design on the current level of abstraction. So we decided to delete the message queues from the design.

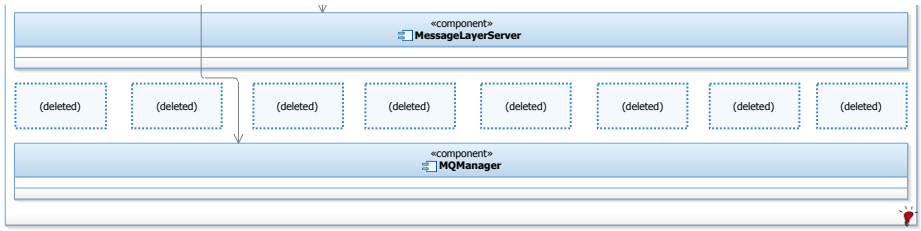


Figure 8.5: The message queues (cf. Fig. 8.4) have been removed by decision 6 and decision 3 is now violated, visible as a marker in the lower-right corner of the package *BusinessProcessLayer*.

Realization. The realization of this decision comprises the deletion of eight components from the business process layer. This can be expressed with an mpatch containing a single generalized model change that describes the deletion of a component. The corresponding element is expressed as a condition-based symbolic reference with the following manually adjusted condition:

self.name.endsWith('Queue') and self.isAbstract=**false** (type: Class)

Applied to the business process layer, this change deletes all eight queues as expected and shown in Fig. 8.5.

However, this modification violates decision 3: *Business Process Engine*, also shown by the violation marker in the design model in Fig. 8.5. The violation is obvious because eight of the added model elements in decision 3 are missing now. Since the deletions are intended, the bindings of the queues can simply be ignored. Figure 8.6 shows a detailed view of the binding, also which and where *Model Element*s are *missing*. We fix the violation by ignoring all bindings of the deleted queues and adding adequate notes (*"The queue has been removed due to decision..."*).

The effort for this realization (creating a generalized model change and applying it to the design) was higher than a manual realization (selecting all queues and deleting them). Fixing the violated binding also took some time. However, this way we already documented right at the binding which model elements were

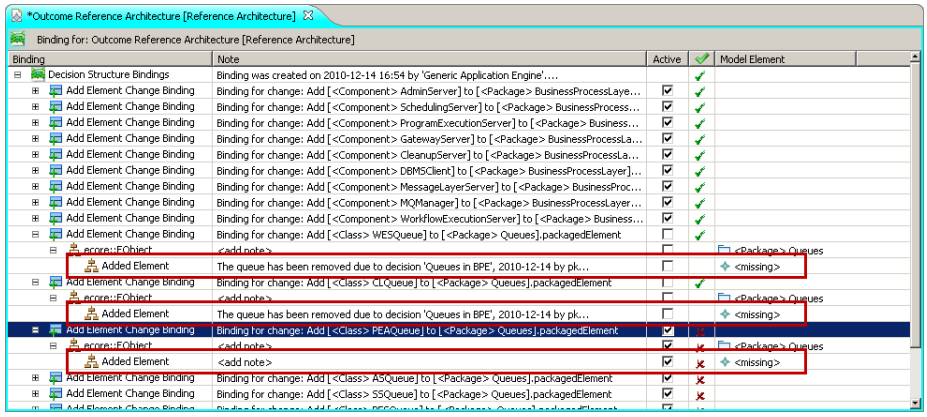


Figure 8.6: We fix the violations from Fig. 8.5 by ignoring all bindings that refer to the eight deleted model elements; the first two have already been ignored and documented in this snapshot, six are still missing.

deleted and, in particular, why. Hence, there is no tool and context switch required for properly documenting why they made particular changes.

8.2.7 Decision 7: Authentication and Security

The two non-functional requirements authentication and security are important for the software system, but they are out of scope in the current state of the design. Moreover, these decisions cannot easily be bound to particular design model elements. Nevertheless, we must keep these issues in mind and, hence, created two open outcomes in order to deal with them later.

8.2.8 Decision 8: Data Modeling

Activities in a business process need data input and also produce data as output for subsequent activities. The input and output data must be specified per activity. We decided to model a uniform way of data input and output with data containers. This is also a proven solution for the chosen reference architecture.

Realization. An abstract class *DataContainer* provides the basic functionality for organizing data in the container, and two concrete data containers are

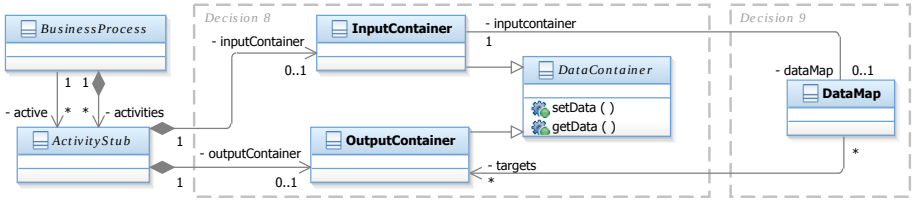


Figure 8.7: The realization of decisions 8 and 9 in the design model.

added to activities. Figure 8.7 shows the newly added elements for this decision (three classes and two composition associations). Like in decision 1, we recorded the model changes and used MPatch to specify a reusable realization specification for that decision. No violations occurred. Again, the effort was the manual modeling plus the MPatch dialogs for the creation and application of the realization.

8.2.9 Decision 9: Data Maps

The data link from the output container of an activity to the input container of other activities must be specified. Available alternatives are data maps as described along with the reference architecture, or direct references between data containers. Our choice are data maps. The rationale behind this decision is to get a clear design and because it is recommended for the chosen reference architecture. The realization went exactly like the one in decision 8.

8.2.10 Decisions 10 and 11: Session Management

The last two decisions concern session management at two different locations in the design model. We skip the detailed description of these two decisions because a similar case has already been presented in the example in Sect. 2.2. Instead, we would like to discuss the scope of decisions.

The BPE requires session management for running business processes, which is also the state of the application (decision 10). This includes the information which activities are active, and what the values of their data containers are. The choice was *Database Session State* applied to the business process layer; the same database as in decision 5 can be used here.

The clients also require session management (decision 11) because several activities acquire data from the web clients that shall be split to multi-page forms. Since the presentation layer shall be as clean as possible and the session handling can easily be handled by clients, our choice is the *Client Session State* alternative.

The point we make here is that the same issue may occur at several places in the design which are independent of each other. Because of this, the outcomes do not relate to each other. Hence, the proposal of subsequent design decisions as discussed in Sect. 6.7 is not globally applicable but it should consider the scope of decisions in the design.

The realizations for both decisions are similar to the one in Sect. 2.2 and comprise several new components, just like decisions 1, 2, 8, and 9. So there is no new insight and we skip a detailed description of these realizations.

8.2.11 Conclusion and Discussion

The domain level test comprises a replay of design decisions of an application for a big telecommunication wholesaler and involves several significant design decisions. We remodeled the architectural design and replayed eleven design decisions based on an experience report [ZDGH05] and the questioning of one of its authors; the real project, however, involved more than 100 decisions. Two of the remodeled design decisions were project-specific, two could not yet be made, the remaining seven could be captured as reusable decisions. The realizations of seven design decisions have been presented in detail, covering different types of decisions and also the violation of previously made decisions in the design model.

The two decisions that could not be captured did not have impact on the design during the investigated period of development. As discussed in the individual decision descriptions, capturing realization specifications was only little overhead because most of them were semi-automatically extracted from the project's design model. Gained benefit is additional documentation that is available via the binding as well as automated validation (goal 1). The latter did, in fact, detect inconsistencies between realizations of decisions; because of a description of the causes, consistency could quickly be restored (goal 3).

This test validated that our tool is applicable to realistic and bigger models than the example in Sect. 2. To be more specific, all but one decision were *existence decisions* (cf. decision ontology in Sect. 3.1.1 on page 18) that added new elements to the design. Their realizations did not violate previous decisions,

in contrast to the decision that modified existing model elements. We learn from the test that our tool supports existence decisions very well with only little extra effort and that inconsistencies produced by evolutional changes in the design (either manual changes or induced by decisions like decision 4) could automatically be identified and easily be fixed.

8.3 Interviews

The intention of the following informal expert interviews is to get feedback from experts about our approach – this is, however, not meant to be an evaluation of our solution. All interviewees are either experts in model-based software development, in making design decisions, or in both. The idea is to let the interviewees compare their current way of documenting, making, and realizing design decisions with the opportunities provided by our tool. The interviews follow the scheme from Appendix D.1, in short:

1. We create a profile of the interviewee.
2. We gather information about the interviewee's current way of documenting, making, realizing design decisions, and keeping the design consistent with made decisions.
3. We present our tool to the interviewee and ask for opinions and feedback (the tool presentation was as neutral as possible to avoid influence on the interviewee's opinion).
4. We ask the interviewee about future directions.

The length of the interviews was between 60 and 95 minutes. A summary of each interview is given in Appendix D.2, here we present and discuss a summary of all interviews and the results. Afterwards we comment on and discuss the results.

8.3.1 The Interviewees

The interviewees are experienced software architects or modelers, both from industry and academia, to get feedback from the research as well as the industrial perspective. Table 8.4 aggregates the profiles of the interviewees in terms of educational degree and experience in UML modeling and decision making. All of them are familiar with UML modeling and have been working in projects

before in which UML was used. Three of the interviewees have ten or more years of practical experience in model-based software development *and* decision making, whereas the other three do not have experience in explicit decision making.

Experience in UML	$4-15$ years	
Experience in decision making	$5-12$ years (and 3 without any experience)	
Highest degree in software engineering	M.Sc.: 4	Ph.D. or higher: 2
Working in	Industry: 3	Academia: 3

Table 8.4: Profile aggregation of all interviewees.

8.3.2 The old Way of Decision Making

We asked the interviewees to describe their current strategy of making, documenting, and realizing design decisions. The idea behind these questions is, on the one hand, to confirm the state of the art in decision making from practitioners and researchers. On the other hand, we can evaluate the feedback concerning our tool: do unexperienced decision makers understand and get along with the proposed tool support for making and realizing design decisions? And are experienced decision makers satisfied with the opportunities offered by the tool?

With one exception, decisions are made ad-hoc, and with two more exceptions, the actual decisions and the rationale behind them are not documented at all. Only the outcomes of the decisions are incorporated into the design and its documentation, but without any traces to the actual decisions. In all interviews, the consistency checks between the design decisions or design documentation and the design artifacts, including design models, was manual work, typically conducted in the form of peer reviews of the design.

8.3.3 Tool Presentation and Feedback

The next part in the interviews was the presentation of our tool. First, we explained the concepts of design decisions including the concepts of issues, alternatives, outcomes, and the idea of automating realizations, too. Second, we demonstrated how to make and realize a design decision (second decision from Sect. 2.2). Third, we modified the design, validated made decisions, and explained how a violated binding can be fixed.

All interviewees understood the idea behind design decisions and their realizations based on model differencing. The tool presentation provoked two-fold reactions.
On the on hand, the automated realization of design decisions as well as the consistency check of the binding were considered useful; "this is a great evolution in software development", said one of the interviewees. The consistency check was further appreciated as a cross-domain validation (decision making and modeling domains).
On the other hand, all interviewees were sceptical and mentioned several success factors for the tool to be usable in practice. The most critical factors are stated below and discussed afterwards.

(1) Most interviewees were concerned about the developers' skills and discipline. Adding automated reuse of design model changes to the tool chain would require proper understanding of the abstract representation of models and model changes, at least for the current GUI (cf. Fig. 7.2 on page 142). Others stated, however, that a concrete syntax or some pretty printer would be a feature that could easily be added on top to provide practical usability. A preview of the changes in the models was suggested to circumvent the complex presentation of model changes. In UML models, for instance, collaborations could be used for visualizing decisions.

(2) Another concern was the support of and flexibility in individual specifications of realizations; generic patterns are of course important, but a higher value was seen in the specification of *known uses*, that are individual, maybe company-specific realizations. Three interviewees said there would be a lot of potential in domains with lots of similar projects, like software product lines[3].

(3) Moreover, it would never be possible to cover the complete design space with such design decisions and their realizations. Some decisions, in fact, influence the design implicitly, for example, by defining coding and modeling guidelines or by prohibiting some design constructs – but this cannot be expressed with the presented concepts for specifying realizations as design model changes.

(4) The last concern we discuss here is tool interoperability. Other tools like bug tracking tools[4], task management systems[5], or requirement management tools[6] are important neighbor systems for assigning tasks, setting priorities, or the like. This is also not supported by the tool at the moment.

[3] See Gomaa [Gom04] for more information about software product lines.
[4] E.g., Bugzilla: http://www.bugzilla.org/
[5] E.g., Mylyn: http://www.eclipse.org/mylyn/
[6] E.g., IBM Rational DOORS [IBM10]

8.3.4 Discussion

All interviewees appreciate the usefulness of automating design decision realizations (goal 2) and consistency checks (goal 3). However, models are not always used and maintained in the projects the interviewees work with, so the approach is not properly applicable in all cases but only if design models are used. Also, the current level of modeling skills is rather low among most developers the interviewees work with; this inhibits chances of a wide adoption of our approach for the time being. We further comment on the four arguments mentioned before.

(1) Developer's skills and discipline: we could address the educational skills of our tool's end-users via coaching and tutorials, and complement the detailed technical view with a concrete syntax and previews of applied model changes directly in the models. This is, however, beyond the scope of an academic prototype.

(2) Individual realizations of decisions: model-independent differences are, in fact, designed to support individual realizations of design decisions and not only predefined patterns and refactorings. The support for several types of design decisions, in particular the distinction between decisions that do or do not explicitly alter the design, is also supported by our tool. However, it is the responsibility of the developers to use it properly and to define and use design decisions adequately.

(3) Design and decision space: our design decision support does not claim to cover the entire design and decision space. It rather supports developers in performing their tasks but it does not substitute thinking. There are many decisions that cannot be expressed as model changes and there is also work on design models that cannot be covered by design decision realizations.

(4) Tool interoperability: the interoperability with other tools is definitely important in practice. However, except for the interaction between a modeling tool and a decision management system, the interoperability with other tools was out of scope for the prototypic implementation of the concepts but should be addressed for productive projects.

8.3.5 Conclusion

To wrap up, the interviewees consider the topic as very important and see a lot of potential in the concepts, especially in goals 2 and 3. But most of them do not see practical relevance for the time being – either because the approach is not applicable to their projects or because of lacking usability. However, the interviewees not knowing about decisions before the interview recognized that they have made numerous unconscious design decision in the past. With one exception, the interviewees said that the presented tool is only usable for experts who know about metamodels and not to the average software developer.

The great benefit was seen in projects within particular domains like product lines. The work by Zimmermann [Zim09] is an example for a domain-specific use of design decisions, namely projects with service oriented architectures.

8.4 Summary

The validation of our approach consists of applicability case studies on refactorings, design patterns, and a replay on the architecture of a real system. This validates the usefulness of our solution for the considered examples. In addition, we interviewed six experts in the UML modeling or software architecting field.

The applicability validation on refactorings has shown that our tool can be used to quickly specify frequently made modifications in design models as design decisions such that they are also applicable to other models – that worked generically for 56% of the tested refactorings. In 24% of the cases, EMF Compare did not detect the modifications properly, and the remaining 20% were not expressible with our tool.

We have similar results with the specification and automated application of design patterns – 34% of them were generically applicable, the application of 60% of them resulted in example elaborations of the patterns in the models and manual work was required afterwards to adjust the patterns to the contexts of the actual models. However, 4% of the patterns could not be expressed with our tool.

All design decisions in the domain level test which altered the design model were expressible with our tool, half of them are potentially reusable, the other half are project-specific. Documented decisions were linked to the design and, hence, were easily accessible (goal 1). Consistency checks were useful for keeping real-

ized design decisions consistent with the actual design models and the binding (goal 3). Some decisions, however, did not immediately affect the design models, hence, they could only be documented but not bound to the design model.

To conclude, most refactorings and design patterns can be expressed as design decisions and the automated application saves manual modeling work. This validates the usefulness of our solution concerning the automation of decision realizations (goal 2). However, in some cases, manual adjustments were required afterwards. The few cases that were not expressible with our tool gave us valuable information about the limitations of our approach. It is worth mentioning that the tool does not necessarily need to be applied to *all* decisions and their realizations; even if only a subset of the documented decisions is bound to the affected design model elements, they can be validated. This is still a benefit in contrast to making, realizing, and validating all design decisions manually.

Furthermore, we got valuable feedback from six software architects and modelers. We asked them to report on their habitual way of documenting, realizing, and validating design decisions and to compare it to the features that our tool offers. Most of the interviewees neither document design decisions at all or they document only their outcomes, mostly without its rationale. All interviewees said that the documentation of design decisions (goal 1) and the consistency checks (goal 3) of the prototype are very useful. The automation of realizations (goal 2) and fixing an inconsistent binding (goal 3), however, require a lot of factors of which most are not given at the moment: lacking developer's education and skills, lacking use of formal models in industry, and a proper integration into the project's tool chain.

Chapter 9

Summary and Conclusion

This thesis presented concepts and tool support for a novel integration of design decision management with the development of model-based software. This chapter summarizes its contents, recapitulates the thesis contributions, and answers the research questions from Chapter 4. Lastly, the outlook on future work lists remaining open issues.

9.1 Summary

This thesis deals with two domains, the decision making domain which originates from the architectural knowledge community, and the domain of model-based software development. There is already sophisticated design decision support for software development in the literature, but only for knowledge management or for decisions linked to source code. There are not many research results yet for design decisions in model-based software development.

An example has been presented to introduce both domains to the reader, decision making and model-based software development, in order to illustrate the basic problems of lacking decision support for design models. Design decisions and design models are created and maintained separately although use cases

like documentation and design decision realization involve both artifacts. This research project aims at integrating both domains by improving design decision support in model-based software development in terms of the following three research questions. The proposed solutions follow afterwards.

1. The first question concerns the documentation of design decisions. Whenever design decisions are made, their rationale is important information for the understandability of the design. How can design decisions be integrated with design models to improve their documentation?

2. The second question concerns the reusability of design decisions and their realizations in design models. If a design decision relates to a recurring design issue, the decision's realization in design models may also recur. Existing tools support recurring design decisions, but how could developers be supported with recurring realizations in design models? How can recurring realizations be automated?

3. The third question concerns the consistency between realized decisions and the design models in which they have been realized. Since the design of a software system evolves over time, realizations of made decisions may be modified and violated. How can consistency between realized decisions and design models be defined, validated, and preserved?

Ad 1: *Improving design decision documentation in model-based software development.*
In the state of the art, the documentation of design decisions is created and maintained separately from design models in model-based software development. An integration only exists in few approaches with code, but not with design models. Consequently, producing and consuming design documentation is a tedious manual task. The proposed metamodel of design decisions comprises a fine-grained integration with design models. A *binding* links decision outcomes to design model elements that are affected by the decision. This way, individual model elements can be traced to design decisions and vice versa.

Ad 2: *Automating recurring decision realizations in design models.*
Some design decisions are frequently made and, thus, their realization in design models is manual, recurring, and error-prone work. Existing approaches for specifying such realizations are either not capable of applying them to arbitrary models, or they require that developers have to learn a new language for the specification of realizations. The proposed model differencing concepts are capable of specifying arbitrary design model changes and support all MOF-based

modeling languages, including UML. Specifications for design decision realizations can be created from example models and generalization algorithms make them applicable to other models.

Ad 3: *Validating consistency between realized decisions and design models.* A design evolves over time, also when design decisions are made and realized. If there are changes made to model elements that realize a particular decision, the design may become inconsistent with that decision. Ensuring consistency between realized design decisions and the actual design is a tedious and time-consuming manual task that is often done in peer reviews. The automated realization of design decisions produces as a by-product a binding between design decision outcomes and affected design model elements. The proposed constraints validate that model changes that were applied as part of decisions' realizations, prevail in the design models. This way, inconsistencies can automatically be detected. Developers can fix such inconsistencies by ignoring inconsistent parts of the binding, by correcting the design model, or by adjusting the binding to the modified design model.

The contributions below realize and validate the aforestated solutions. Contrary to the solutions, the contributions are logically combined to independent parts: concepts for model differences, concepts for design decision support, and the validation results via the prototypic implementation.

Model-independent Differences. The conceptual solution for specifying and automating realizations of design decisions is called model-independent differences. It allows to distill model differences from two versions of a model such that changes made to that model are also applicable to other models. The term model independence refers, on the one hand, to the property that calculated differences are self-contained and do not need the compared models in order to be applicable to other models. On the other hand, it refers to the fact that calculated changes are applicable to arbitrary models, also models that were created independently of the compared models. These concepts are used in the other two contributions.

Design Decision Support. This project complements existing concepts for capturing, identifying, making, realizing, and validating design decisions with a conceptual solution for design decision support in model-based software development. It uses model-independent differences for realization specifications of reusable design decisions which can be created by developers from exemplary models or from models of other projects. They can then be used to interac-

tively apply realizations of design decisions to design models. This provides an easy way for developers to document design decisions and to automate recurring work. Moreover, a binding links decision outcomes to affected design model elements. This binding can be validated to ensure consistency between realized design decisions and design models.

This contribution meets the goals of all three research questions. Links between design decisions and affected design model elements allow a navigation between both artifacts; this meets the first goal concerning decision documentation. Using model-independent differences for the specification and application of realization specifications of reusable design decisions meets the second goal concerning automation of decision realizations. The validation mechanism meets the third goal concerning consistency.

Tool Support and Validation. A prototypic tool has been developed that implements the aforementioned concepts. The implementation of model-independent differences is named *MPatch* and is contributed to the open source project Eclipse as a generic tool for creating and applying model differences. The implementation of design decision support is flexible and works with different decision management systems. An interface has been presented that allows any decision management system to be used with the proposed solution which implements that interface. The ADK Web Tool [Zim09] and the DTU Decision Server (cf. Sect. 7.5) are supported so far.

The tool has been used to validate the technical feasibility and applicability of the proposed solution with design patterns and refactorings as design decisions. Moreover, a replay of several design decisions of a real project based on an experience report validates the applicability on realistic models. Lastly, interviews of six experts in model-based software development, in making design decisions, or in both provided valuable feedback about the tool, its concepts, and success factors for its use in practice.

This contribution shows the technical feasibility of all three goals and validated the solution's usefulness and applicability to realistic examples.

9.2 Conclusion

The conclusion of this thesis discusses which claims of the solution are met and which are not supported but future work.

Design Decisions Documentation. The documentation of design decisions in model-based software development is improved by this solution because developers do not have to switch tools anymore when working on design models and documenting design decisions. The basic functionality for documenting and maintaining design decisions can now be done from within the modeling tool because of a link between design decisions and affected design model elements. Tool supported navigation between both artifacts is also possible now. The usefulness of such an improved navigation of documented decisions is not yet evaluated in realistic settings, e.g. in a running project; this is future work.

Automation of Design Decision Realizations. The technical feasibility of model-independent differences has been proven with test cases for three different types of models. The capability of generalizing model differences has been validated with practical examples, namely design patterns and refactorings: approx. half of the captured realizations were generically applicable without additional effort. Another 40% are generically applicable with manual adjustments after the realizations have been applied. The remaining 10% of the cases could not be expressed with model-independent differences, but realizing them manually is still compatible with the proposed solution. Furthermore, replaying the design of a realistic example has shown that the solution is also applicable to bigger models with realistic properties. However, an evaluation in a running project could not be performed due to the limited time frame of this project. In the long term, an evaluation in a running project should include conditions like time pressure, distributed development, and coordination and collaboration within the development team.

Consistency between Design Decisions and Design Models. With the proposed solution, automated consistency validation is finally possible. Different ways of visualizing violations and three opportunities of fixing inconsistencies are sufficient for all presented examples. A technical limitation is currently the validation of deletions in a model – non-existence of model elements cannot easily be checked. Moreover, manually fixing a violated binding is tedious and could further be improved by tool support. An evaluation of the consistency validation under realistic conditions is again future work.

9.3 Future Work

The concepts and tools developed in this project have successfully been applied to the investigated examples. However, several interesting aspects have not yet

been considered and are worth further investigation and research. The remaining open issues are listed below concerning conceptual work and tool support.

Conceptual Future Work. Using model-independent differences for capturing and describing design patterns and refactorings raised the need for a yet more flexible representation of model changes. In particular, a dynamic specification of parameters during application-time of model changes is missing for the time being; now it is only possible to apply example elaborations and these examples must be customized afterwards.

An evolving design may violate bindings which must be corrected manually by the developer. Existing approaches for traceability management could be integrated to update and adjust the binding automatically in some cases where possible. The difficulty is here to preserve the semantics of the bindings.

The evaluation of the solution must still be complemented with a case study in which the tool is applied in a running software development project and not only after the project has been finished. Only such an experiment can answer the questions whether the concepts work for documenting, capturing, realizing, and validating design decisions under time pressure and with coordination and collaboration among the team members.

Future Work on Tool Support. A practical application of the tool requires a proper integration into the development process and possibly the interoperability with other tools like requirement management tools. Moreover, a missing and remaining component in the tool is the implementation of decision guidance support by proposing subsequent decisions. Conceptual ideas have been presented but without a supporting tool to validate the ideas.

Usability is very important for using the tool in practice but the visualization of model-independent differences is probably too technical for most developers. A pretty printer or previews of the models to which the differences shall be applied could solve that issue. Also, the format and style of violation messages was considered too technical and complicated for most developers.

In the end, we have to convince developers to think about the decisions they make and that they realize the importance and benefits of properly documented design decisions and rationale. The advantage of capturing design decisions and their realizations is not only an improvement of the design documentation, but now it also provides a method for automating recurring work and improving software quality by validating made design decisions.

Appendix A

Transformation Specification from emfdiff to indepdiff

This section contains the complete transformation specification from model-dependent differences of EMF Compare to model-independent differences as explained in Sect. 5.6.3. The complete metamodel of EMF Compare, the source metamodel of the transformation, is shown in Fig. A.1, the complete metamodel of model-independent differences, the target metamodel of the transformation, is shown in Fig. A.2.

In QVT Operational Mappings, a transformation operates on *in*-, *out*-, and *in-out*-models, which means they are treated as input, output, or both. Listing A.1 shows the signature of the transformation. *EmfDiff* (short form for instances of the EMF Compare metamodel) is used as input, *MPatch* as output (line 6). The main mapping (line 8) retrieves all instances of *DiffModel* and explicitly calls the mapping *toMPatchModel*. *log*-commands (lines 9 and 11) are used for debugging purposes.

```
1 import org.eclipse.emf.compare.mpatch.emfdiff2mpatch.lib.mpatchlibrary;
2
3 modeltype EmfDiff
4         uses diff('http://www.eclipse.org/emf/compare/diff/1.1');
5 modeltype MPatch
6         uses mpatch('http://www.eclipse.org/emf/compare/mpatch/1.0');
7
8 transformation emfdiff2mpatch(in emfdiff : EmfDiff, out mpatch : MPatch);
9
```

```
10 main() {
11     log('starting transformation...', emfdiff);
12     emfdiff.objectsOfType(DiffModel)->map toMPatchModel();
13     log('transformation finished!', this);
14 }
```

Listing A.1: Signature and entry point of the transformation

Root Mapping. Each transformation starts with the mapping *toMPatchModel* which maps the *DiffModel* of the source metamodel to *MPatchModel* of the target metamodel (line 1 in Listing A.2). This rule is in particular responsible for setting up the target model; it also stores the URI of the original models (lines 6–10).

```
1  mapping EmfDiff::DiffModel::toMPatchModel() : IndepDiff::MPatchModel {
2      init {
3          -- lets check whether we got a diff from two models
4          assert fatal (self.leftRoots->notEmpty() and
5                  self.rightRoots->notEmpty())
6              with log('leftRoots or rightRoots is empty!', self);
7          assert error (self.ancestorRoots->isEmpty())
8              with log('ancestorRoots must be empty!', self);
9
10         -- a bag containing all relevant DiffElements:
11         var elements := self.ownedElements.allInstances(DiffElement).
12             oclAsType(DiffElement)->reject(oclIsTypeOf(DiffGroup))
13     }
14     oldModel := self.rightRoots->iterate(eobj; uris : String = '' |
15         uris.concat(eobj.eResource().toUriString()).concat(' '));
16     newModel := self.leftRoots->iterate(eobj; uris : String = '' |
17         uris.concat(eobj.eResource().toUriString()).concat(' '));
18     result.source := self.oclAsType(EObject).eResource().toUriString();
19
20     changes := elements->map toIndepChange()->asOrderedSet()
21 }
```

Listing A.2: Root mapping: *toMPatchModel*.

This mapping iterates over all concrete changes of the input model and calls another rule for each of them. More precisely, all instances of *DiffElement* (only subclasses of it because *DiffElement* is abstract) are collected, regardless where they are contained in the source model, using the OCL construct *allInstances* (lines 3–4). Groups are, however, filtered because they do not describe any change in the model (line 4). Then the mapping *toIndepChange* is called (line 12). The result is finally transformed into an ordered set and assigned to the property *changes* of an implicitly created object of type *MPatchModel*; the signature in line 1 specifies that an object of that type will be created during the initialization of this mapping.

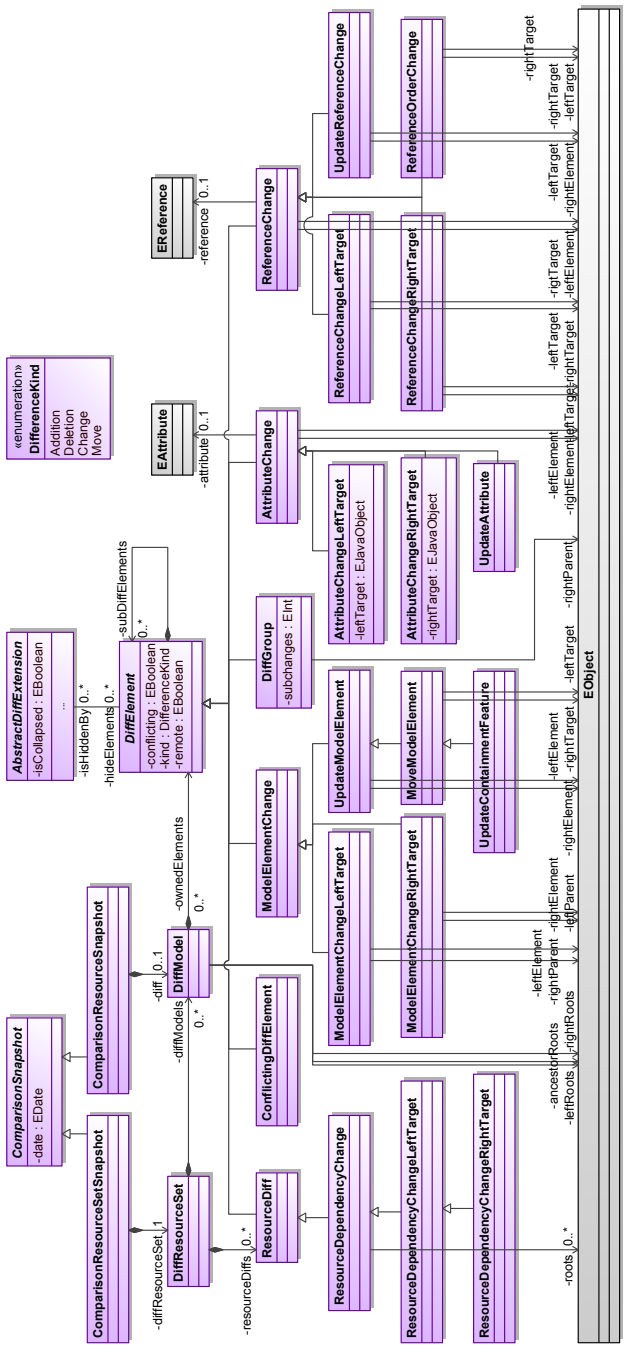


Figure A.1: The complete metamodel of EMF Compare.

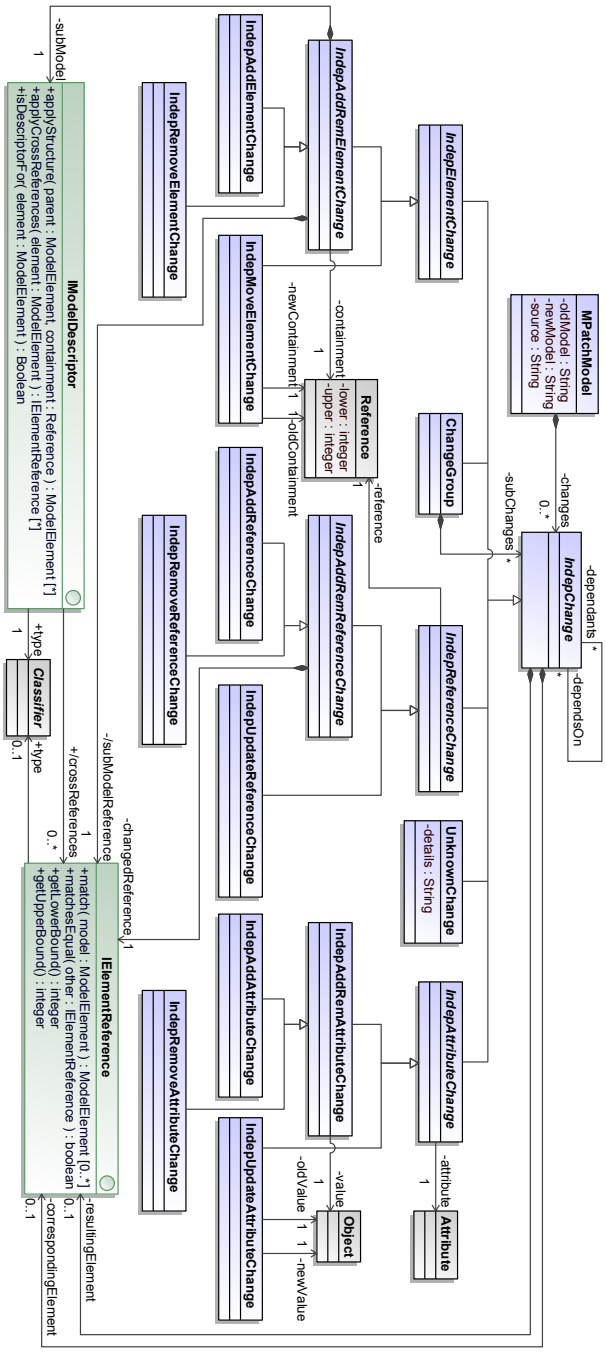


Figure A.2: The complete metamodel of model-independent differences.

Overloaded Mapping. The mapping *toIndepChange*, that is called in line 12 of Listing A.2, is an overloaded mapping. The type of the source element for which the mapping is called, specifies during execution time of the transformation which overloaded mapping is called (cf. Tab. 5.10 on page 71). *toIndepChange* is also defined for the abstract source type *DiffElement* for two reasons (cf. Listing A.3). First, all other overloaded mappings inherit from the former; this way, additional code might be added here that is executed for all change types (lines 9–10). Second, if there is no mapping defined for a particular change type (lines 5–7), the variable *result* has not been initialized yet; the mapping *toUnknownChange* is called in that case to process the unknown change type.

```
1 -- mapping for abstract DiffElement:
2 -- If DiffElement is of a known type, a concrete mapping created the
3 -- 'result' object. Otherwise, the unknown mapping will be called.
4 mapping EmfDiff::DiffElement::toIndepChange() : IndepDiff::IndepChange {
5     init {
6         -- check whether the result is already initialized, i.e. the
7         -- mapping operates on a known sub type;
8         -- if not, it is an unknown change type.
9         if result = null or result.oclIsUndefined() then
10            result := self.map toUnknownChange()
11        endif
12    }
13    -- the code below is executed for all change types.
14    log('source diff element processed', this)
15 }
```

Listing A.3: Overloaded mapping: *toIndepChange*.

Mapping Unknown Changes. Changes are transformed into unknown changes whenever there is no suitable concrete mapping. Listing A.4 lists that mapping; it stores some information (via the operation *repr()*) about the unsupported change such that the user gets an idea of which change is not supported.

```
1 -- mapping for all unknown change types
2 mapping DiffElement::toUnknownChange() : mpatch::UnknownChange {
3     info := '(' + self.metaClassName() + '): "' + self.repr() + '"';
4 }
```

Listing A.4: Mapping for unsupported changes: *toUnknownChange*.

Concrete Mappings. The concrete mappings for the individual change types are very similar to each other. As an example, we explain the mapping for added elements in Listing A.5, because it covers all relevant concepts. The keyword *inherits* indicates the overridden mapping, defined for the concrete type *ModelElementChangeLeftTarget*. It explicitly initializes the resulting object of type *IndepAddElementChange* in line 4 and sets all relevant properties. The corresponding element is set to the left parent in line 5; we have seen

both references (*correspondingElement* and *leftParent*) already in the metamodel excerpts. However, the operation *toSymbolicReference* is not defined in this transformation specification but in a separate library; the same applies to *toModelDescriptor* in line 7. Both operations require access to reflective operations of the models and perform complex String operations. The former is not possible within the transformation language, the latter is much easier in other programming languages like Java. Therefore, a blackbox library providing these operations is presented in the following section. The import statement for the library is stated in line 1 in Listing A.1.

```
1  mapping ModelElementChangeRightTarget::toIndepChange() : IndepChange
2          inherits DiffElement::toIndepChange {
3      init {
4          result := object IndepAddElementChange {
5              correspondingElement :=
6                      self.leftParent.toSymbolicReference();
7              resultingElement := self.rightParent.toSymbolicReference();
8              containment := self.rightElement.eContainmentFeature();
9              subModel := self.rightElement.toModelDescriptor()
10         }
11     }
12 }
```

Listing A.5: Mapping *toIndepChange* for the change type describing added elements.

All remaining mappings are printed in Listing A.6.

```
1  -- mapping for moved elements
2  mapping MoveModelElement::toIndepChange() : IndepChange
3          inherits DiffElement::toIndepChange {
4      init {
5          result := object IndepMoveElementChange {
6              correspondingElement :=
7                      self.rightElement.toSymbolicReference();
8              resultingElement := self.leftElement.toSymbolicReference();
9              oldContainment := self.rightElement.eContainmentFeature();
10             newContainment := self.leftElement.eContainmentFeature();
11
12             -- get old and new parents from both models
13             oldParent := self.rightElement.eContainer().
14                     toSymbolicReference();
15             newParent := self.leftElement.eContainer().
16                     toSymbolicReference();
17         }
18     }
19 }
20
21 -- mapping for removed elements
22 mapping ModelElementChangeRightTarget::toIndepChange() : IndepChange
23         inherits DiffElement::toIndepChange {
24     init {
25         result := object IndepRemoveElementChange {
26             correspondingElement :=
27                 self.rightElement.eContainer().toSymbolicReference();
28             resultingElement := self.leftParent.toSymbolicReference();
29             containment := self.rightElement.eContainmentFeature();
30
31             -- this model descriptor is intended to store a sub-model
```

```
32              -- which can be re-build later on.
33              subModel := self.rightElement.toModelDescriptor(true);
34          }
35      }
36  }
37
38  -- mapping for changed attributes (cardinality of this attribute is 1)
39  mapping UpdateAttribute::toIndepChange() : IndepChange
40          inherits DiffElement::toIndepChange {
41      init {
42          result := object IndepUpdateAttributeChange {
43              changedAttribute := self.attribute;
44              correspondingElement :=
45                  self.rightElement.toSymbolicReference();
46              resultingElement := self.leftElement.toSymbolicReference();
47              oldValue := self.rightElement.eGet(self.attribute);
48              newValue := self.leftElement.eGet(self.attribute);
49          }
50      }
51  }
52
53  -- mapping for removed attributs (cardinality of this attribute is >1)
54  mapping AttributeChangeRightTarget::toIndepChange() : IndepChange
55          inherits DiffElement::toIndepChange {
56      init {
57          result := object IndepRemoveAttributeChange {
58              changedAttribute := self.attribute;
59              correspondingElement :=
60                  self.rightElement.toSymbolicReference();
61              resultingElement := self.leftElement.toSymbolicReference();
62              value := self.rightTarget;
63          }
64      }
65  }
66
67  -- mapping for added attributes (cardinality of this attribute is >1)
68  mapping AttributeChangeLeftTarget::toIndepChange() : IndepChange
69          inherits DiffElement::toIndepChange {
70      init {
71          result := object IndepAddAttributeChange {
72              changedAttribute := self.attribute;
73              correspondingElement :=
74                  self.rightElement.toSymbolicReference();
75              resultingElement := self.leftElement.toSymbolicReference();
76              value := self.leftTarget;
77          }
78      }
79  }
80
81  -- mapping for a changed reference (cardinality of this reference is 1)
82  mapping UpdateReference::toIndepChange() : IndepChange
83          inherits DiffElement::toIndepChange {
84      init {
85          result := object IndepUpdateReferenceChange {
86              reference := self.reference;
87              correspondingElement :=
88                  self.rightElement.toSymbolicReference();
89              resultingElement := self.leftElement.toSymbolicReference();
90
91              -- check the model whether the reference is null
92              var rightTarget := self.rightElement.eGet(reference);
93              var leftTarget := self.leftElement.eGet(reference);
94              oldReference :=
95                  if not (rightTarget.oclIsUndefined()) then
96                      rightTarget.oclAsType(EObject).toSymbolicReference()
```

```
97                  else null endif;
98              newReference :=
99                  if not (leftTarget.oclIsUndefined()) then
100                     leftTarget.oclAsType(EObject).toSymbolicReference()
101                 else null endif
102         }
103     }
104 }
105
106 -- mapping for removed references (cardinality of this reference is >1)
107 mapping ReferenceChangeRightTarget::toIndepChange() : IndepChange
108         inherits DiffElement::toIndepChange {
109     init {
110         result := object IndepRemoveReferenceChange {
111             reference := self.reference;
112             correspondingElement :=
113                 self.rightElement.toSymbolicReference();
114             resultingElement := self.leftElement.toSymbolicReference();
115             changedReference := self.rightTarget.toSymbolicReference();
116         }
117     }
118 }
119
120 -- mapping for added references (cardinality of this reference is >1)
121 mapping ReferenceChangeLeftTarget::toIndepChange() : IndepChange
122         inherits DiffElement::toIndepChange {
123     init {
124         result := object IndepAddReferenceChange {
125             reference := self.reference;
126             correspondingElement :=
127                 self.rightElement.toSymbolicReference();
128             resultingElement := self.leftElement.toSymbolicReference();
129             changedReference := self.leftTarget.toSymbolicReference();
130         }
131     }
132 }
```

Listing A.6: The mappings *toIndepChange* for all remaingin change types.

Appendix B

Similarity Algorithm for Strings

The *Scope Expansion* transformation explained in Sect. 5.7.1 uses the algorithm in Listing B.1 to calculate the similarity between two strings. It is based on two rules:

Rule 1: string containment is of high importance.
Rule 2: case sensitivity is not important but should neither be neglected.

The algorithm takes the two strings and a threshold as parameters. In practice, a threshold below 0.5 is rarely useful. Because of this, we added the condition in line 18 to optimize the algorithm and avoid the expensive call to the Levenshtein distance.

```
1  isSimilar(str1, str2, threshold): boolean
2
3  // let str2 be the longer string
4  if (str1.length > str2.length)
5    swap(str1, str2)
6
7  // equality and containment are important for us
8  if (str2 equals str1)
9    similarity = 1
10 else if (str2 contains str1)
11   similarity = 0.5 + 1.25 * str1.length /
12                (str2.length + 2 * str1.length)
13 else if (str2.lowerCase equals str1.lowerCase)
14   similarity = 0.9
15 else if (str2.lowerCase contains str1.lowerCase)
16   similarity = 0.4 + 1.25 * str1.length /
17                (str2.length + 2 * str1.length)
18 else if (threshold > 0.5)
19   similarity = 0
20 else
21   // use the levenshtein distance in all other cases
22   distance = LevenshteinDistance(str1, str2)
23   similarity = (1 - distance / str2.length) / 2
24
25 return similarity >= threshold
```

Listing B.1: A similarity algorithm for strings in pseudo code.

Appendix C

Binding Metamodel and Constraints

The binding metamodel has been partially explained in Sect. 5.9. The entire metamodel is given in Fig. C.1. Except for attribute changes, which are all bound by the same change binding in the metamodel (*AttributeChangeBinding*), each concrete change binding corresponds to exactly one of the concrete change types. Each concrete change binding refers to all model elements that are affected by a change. Table C.1 lists an overview of all affected model elements for each change type.

A special case is the *AddElementChangeBinding*, because it requires a specific binding for added model elements described by model descriptors: a *SubModelBinding* has references to two model elements and contains further element bindings. The two references are the parent and the actually added element. Further element bindings are bindings for sub-elements and cross references.

Section 6.5 listed only three examples for constraints, one for the element level, one for the change level, and one for the decision level. The following is a complete list of all element and change level constraints; table C.2 gives an overview grouped by change type. If the constraints are valid, the respective change prevails in the model. However, there are no constraints for changes describing deleted elements or reference, because there is no information in the binding possible which specifies which element was deleted (because it does not

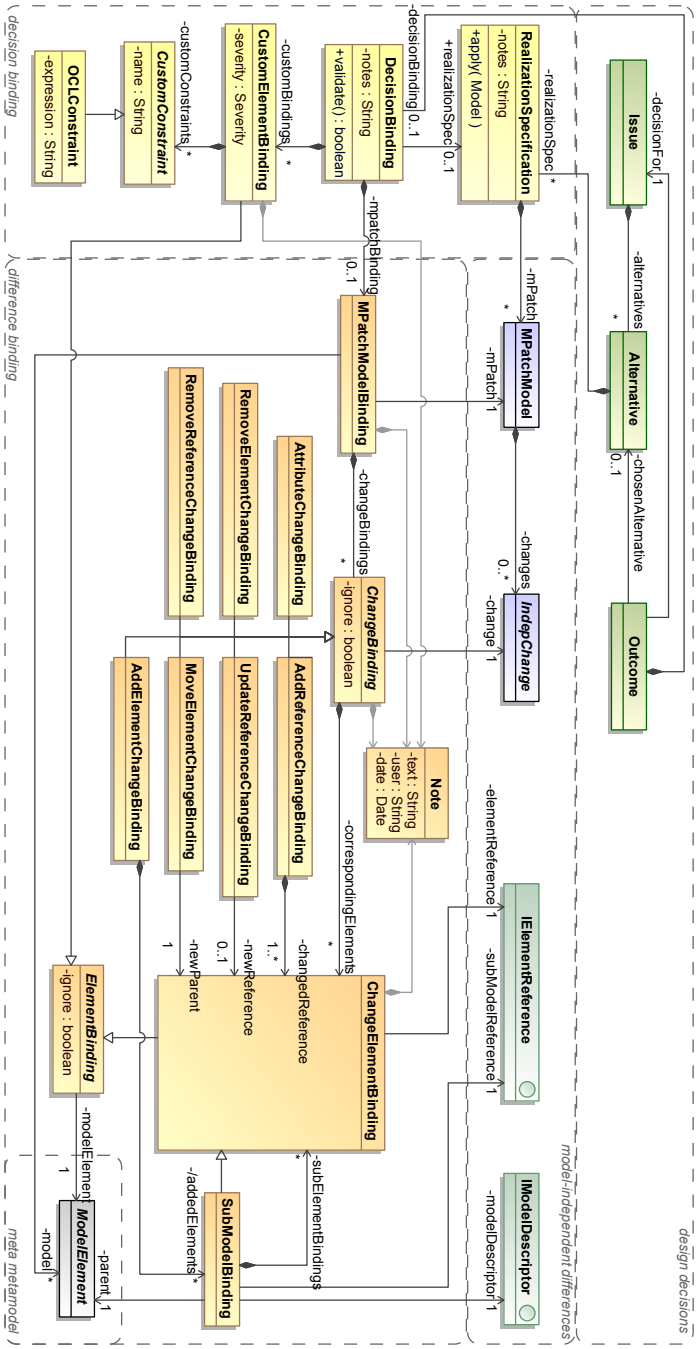


Figure C.1: The complete metamodel of the binding between design decisions and design models.

Change	Model Reference	Meaning
Add element change	• *correspondingElement*	parent
	• *subModelReference*	added element
	• *crossReferences*	cross references
Remove element change	• *correspondingElement*	parent
	• *subModelReference*	deleted element
Move element change	• *correspondingElement*	moved element
	• *oldParent*	old container
	• *newParent*	new container
Add reference change	• *correspondingElement*	reference owner
	• *changedReference*	reference target
Delete element change	• *correspondingElement*	reference owner
	• *changedReference*	reference target
Update reference change	• *correspondingElement*	reference owner
	• *oldReference*	old target
	• *newReference*	new target
Add attribute change Remove attribute change Update attribute change	• *correspondingElement*	attribute owner

Table C.1: An overview of all affected model elements for each change type.

exist anymore) or which element is not referenced anymore (it may also not exist anymore).

This list is actually independent of any decision but applies to the decision binding only. The constraints are stated as invariants in the Object Constraint Language (OCL).

```
1  context ElementBinding
2  ----------------------
3  -- check that the referenced elements exists
4  inv Model_elements_exist:
5      self.ignore or (not self.modelElement.oclIsUndefined())
6
7  context ChangeBinding
8  ---------------------
9  -- check bounds of corresponding elements
10 inv Lower_bound_corresponding_elements:
11     self.ignore or self.correspondingElements->size() >=
12         change.correspondingElement.getLowerBound()
13 inv Upper_bound_corresponding_elements:
14     self.ignore or self.change.correspondingElement.upperBound < 0 or
15         self.correspondingElements->size() <=
16             self.change.correspondingElement.getUpperBound()
17
18 context MPatchModelBinding
19 -- check that the referenced model exists and it is not a proxy
20 inv Model_exist_ERROR:
21     not model.oclIsUndefined() and not model.eIsProxy()
```

Listing C.1: Model Exists Constraint

Change	Constraint name	Severity	Listing
—	Model exists	error	C.1
Independent of any type	Model elements exist	error	C.2
Applies to all types	Bounds of corresponding elements	error	C.3
Add Element Change	Added Element Exists	error	C.4
Add Element Change	Added Element is Child of expected parent	warning	C.4
Move Element Change	Moved Element is Child of expected parent	error	C.5
Add Reference Change	Added Reference prevails	error	C.6
Add Reference Change	Cardinality of Added Reference is correct	warning	C.6
Update Reference Change	Updated Reference prevails	error	C.7
Update Attribute Change	Updated Attribute prevails	error	C.8
Add Attribute Change	Added Attribute exists	error	C.8
Remove Attribute Change	Removed Attribute does not exist	error	C.8

Table C.2: List of all constraints grouped by changes types.

```
1 context ElementBinding
2 -- check that the referenced elements exist and they are not proxies
3 inv Model_elements_exist_ERROR:
4     ignore or (not modelElement.oclIsUndefined() and
5         not modelElement.eIsProxy())
```

Listing C.2: Model Element Exists Constraint

```
1 context ChangeBinding
2 -- check that the cardinality of the corresponding elements matches the
3 -- actual number of bound model elements
4 inv Lower_bound_corresponding_elements_ERROR:
5     correspondingElements->size() >=
6         change.correspondingElement.lowerBound
7 inv Upper_bound_corresponding_elements_ERROR:
8     change.correspondingElement.upperBound < 0 or
9         correspondingElements->size() <=
10            change.correspondingElement.upperBound
```

Listing C.3: Bounds of Corresponding Elements Constraints

```
1 context SubModelBinding
2 -- check that the referenced element exists and it is not a proxy
3 inv Self_element_exist_ERROR:
4     ignore or (not selfElement.oclIsUndefined() and
5         not selfElement.eIsProxy())
6 -- added model element exists at the expected parent
7 inv Added_model_element_is_child_of_correct_element_WARNING:
8     ignore or (modelElement = selfElement.eContainer())
```

Listing C.4: Added Element Constraints

```
1  context MoveElementChangeBinding
2  -- make sure that the moved element is really at the new place
3  inv Moved_model_elements_are_child_of_new_parent_ERROR :
4      correspondingElements->forAll(
5          ignore or (modelElement.eContainer() = newParent.modelElement))
```

Listing C.5: Moved Element Constraint

```
1  context AddReferenceChangeBinding
2  -- the added reference must prevail
3  inv Added_reference_prevails_ERROR:
4      correspondingElements->forAll(ignore or (modelElement.
5          eGetAsSequence(change.oclAsType(mpatch::IndepReferenceChange).
6              reference)->asSet()->collect(oclAsType(ecore::EObject))->
7                  includesAll(changedReference->collect(modelElement))))
8  -- check that the cardinality of the added references matches the actual
9  -- number of bound model elements
10 inv Lower_bound_Add_References_ERROR:
11     changedReference->size() >= change.
12         oclAsType(mpatch::IndepReferenceChange).reference.lowerBound
13 inv Upper_bound_Add_References_ERROR:
14     change.oclAsType(mpatch::IndepReferenceChange).reference.
15         upperBound >= changedReference->size() or
16             change.oclAsType(mpatch::IndepReferenceChange).reference.
17                 upperBound < 0
```

Listing C.6: Added Reference Constraints

```
1  context UpdateReferenceChangeBinding
2  -- the new reference must prevail
3  inv NewReference_must_prevail_ERROR:
4      (not newReference->isEmpty()) implies
5          correspondingElements->forAll(ignore or
6              (modelElement.eGet(change.oclAsType(
7                  mpatch::IndepReferenceChange).reference) =
8                      newReference.modelElement))
```

Listing C.7: Update Reference Constraint

```
1  context AttributeChangeBinding
2  -- helper variable
3  let c: IndepAttributeChange =
4      if change.oclIsTypeOf(mpatch::IndepAttributeChange) then
5          change.oclAsType(mpatch::IndepAttributeChange)
6      else null endif
7  -- changed attribute must prevail
8  inv ChangedAttribute_must_prevail_ERROR:
9      change.oclIsTypeOf(mpatch::IndepUpdateAttributeChange) implies
10         correspondingElements->forAll(ignore or
11             (modelElement.eGet(c.changedAttribute) =
12                 change.oclAsType(mpatch::IndepUpdateAttributeChange).
13                     newValue))
14 -- added attribute must prevail
15 inv AddedAttribute_must_prevail_ERROR:
16     change.oclIsTypeOf(mpatch::IndepAddAttributeChange) implies
17         correspondingElements->forAll(ignore or
18             (modelElement.eGetAsSequence(c.changedAttribute)->includes(
19                 change.oclAsType(mpatch::IndepAddRemAttributeChange).
20                     value)))
21 -- removed attribute must not prevail
22 inv RemovedAttribute_must_not_prevail_ERROR:
```

```
23      change.oclIsTypeOf(mpatch::IndepRemoveAttributeChange) implies
24          correspondingElements->forAll(ignore or
25              (modelElement.eGetAsSequence(c.changedAttribute)->excludes(
26                  change.oclAsType(mpatch::IndepAddRemAttributeChange).
27                      value)))
```

Listing C.8: Attribute Constraints

Appendix D

Interviews

Section 8.3 discusses interviews in which our tool was presented to experts. All interviews followed the scheme shown in Sect. D.1. Statements in *italics* give additional information to particular parts of the interview. The individual interview results are listed in Sect. D.2.

D.1 Interview Scheme

Title:
'Design Decision Support in Model-based Software Development'

This interview is part of the Ph.D. project of Patrick Könemann.
Its purpose is to get feedback for the prototypic tool.

[ ] I, the interviewee, agree that the interview is anonymous and that it is recorded (audio); the recorded data will only be used for evaluating the interview and will be deleted after the evaluation is complete but latest 3 months after the interview.

Part 1: Assessment of the Interviewee

a) Educational degree (e.g. M.Sc., Ph.D., ...)
b) Role in project (e.g. software architect, modeler, programmer, ...)
c) Self-assessment: please rate your experience in the following fields
   - UML modeling experience in years
   - UML modeling skills
   - 'Decision making' experience in years

Part 2: The old Way

The interviewee should freely talk about her/his best practices, experiences and good and bad examples and projects.

What is your typical way of...
a) making design decisions? (general procedure, process)
b) looking up made decisions / documentation?
c) realizing decisions in the design (model)?
d) keeping your design (model) consistent with decisions / documentation / requirements?

Part 3: Intention of this Research Project

a) Integrated documentation
Tool demo: Design decisions in the modeling tool – Issues, Alternatives, Outcomes, and Realizations (showing decision 2 from Sect. 2.2).
What is the interviewee's opinion? How important is that information in practice?

b) Automation of recurring work
Tool demo: Recurring changes in the design model are captured as best practices and can be reused (applying decision 2 from Sect. 2.2).
What is the interviewee's opinion? How useful is that in practice?

c) Consistency
Tool demo: Automatic consistency checks between design models and made design decisions; Model evolution, violations, and correcting the binding (following the example in Sect. 6.6.3).
What is the interviewee's opinion? How useful is that, how could that be used in practice?

d) Overall Feedback
What is the interviewee's general opinion?
How could design decision support be integrated into your development projects?
Could you comment on the applicability to your development projects?
Could you comment on other potential problems (technical problems, tool chain integration, education of employees)?

Part 4: Opinions about Design Decisions in the Future

a) How important is the visualization of decisions, their relations, and their integration with design models?
b) How do you see the impact of decisions in software development and software understandability?
c) How important is the degree of automation?
d) How important is automated validation of decisions and models? E.g. during software maintenance?

Thank you very much for the interview.

D.2 Interview Summaries

D.2.1 Interview 1

Part 1: Assessment of the Interviewee

Education	Ph.D.
Roles	IT-Architect
UML experience	12 years
UML skills	9/10
Decision making experience	12 years

Before 2005, the interviewee was an IT architect in industry for 16 years. Since 2005, the interviewee is active in the research community for architectural design and architectural decisions.

Part 2: The old Way

a) *What is your typical way of making design decisions? (general procedure, process)*
Until 2005:

- Architectural decisions are rather cross-cutting concerns
- Successful story of making decisions in meetings and workshops (2002–2005):
  - Decisions and potential solutions prepared before meetings
  - Explicit documentation of decision, rationale, and alternatives after meetings
- Template in IT Architect community, e.g. wiki or MS World/Excel
- Explicit section or appendix in architectural design documentation, e.g. as bullet point list
- Interviewee learned retrospective decision documentation (after they have been made)
- Proactive preparation of decisions in a particular domain that are about to be made turned out to be very practical

After 2005:

- Preparation of decisions with reusable assets / guidance models per domain (e.g. architectural style) has been proven to be very useful.

b) *What is your typical way of looking up made decisions / documentation?*

- Experiences in case of extending legacy designs
- Documents possibly have a bad quality and might not reflect the current design.
- First step: study of existing documents, e.g. requirement and design documents.
- Second step: identification of specific architectural artifacts, depending on the software process.
- Third step: prepare questionnaire for knowledge carriers.
  Get at least 2–3 opinions.

c) *What is your typical way of realizing decisions in the design (model)?*
(the same before and after 2005)

- Depends on decision type!
  Out of scope here: business or logistic decisions.
- Examples of made decisions: which architectural styles and patters are used, which pattern languages, but also which coding guidelines shall be used and what is the package structure for the code.
- Examples of realizing a decision or delegating it to developers:
  - architect writes sample code/project that shows the developers how the decision shall be realized (also called 'architectural template')
  - Informal text describing the realization or link to reference implementation
  - Coaching developers and reviewers

d) *What is your typical way of keeping your design (model) consistent with decisions / documentation / requirements?*
Until 2005:

- Good point, projects may fail because of that.
- Code review! Ask developers and do not believe everything what they say in the first place.
- Compare code and running system with design specification.
- This is all manual work!

Part 3: Intention of this Research Project

a) Integrated documentation
Tool demo: Design decisions in the modeling tool – Issues, Alternatives, Outcomes, and Realizations.
What is the interviewee's opinion? How important is that information in practice?

- Concepts are known: Problem, potential solutions, chosen solution.
- Reuse of realizations sounds promising, with some remarks:
  - It must be possible to specify different realizations for the same alternative / pattern.

  - A particular solution may occur several times in a project. A particular solution may also contain sub-patterns down to atomic blocks. There might be an n:m dependency between realizations and these atomic block.
    Example: a UML package might be affected in multiple decisions and a decision might affect multiple model elements like a package and containing classes.
  - Not all decisions may be represented like that in the design, some decisions may affect the design but cannot be linked to concrete design element.
    For example the choice of a reference architecture.
  - It is not possible to cover the complete decision space with such realizations.
- Important is the consideration of both tracks, the design space and the decision space, and not neglecting either of them.

b) Automation of recurring work
Tool demo: Recurring changes in the design model are captured as best practices and can be reused.
What is the interviewee's opinion? How useful is that in practice?

- Each context switch is mental work, even tough it is just two mouse click to switch between two programs.
- Every approach is welcome that moves these two domains (decision making and working on the design) closer together.
- Typical quality attributes apply to such tools, such as scalability.
- Naming is very important and probably not easy to handle:
  Different people in the team may use different terms for the same concepts.
- Patterns might change!
  Design templates, the realizations, might change, for examples, when it did not work properly in one place.
- Concerning the tradeoff 'manually realizing the solution in the design' vs. 'refining the mapping during automatic application of predefined realizations':
  Tool analogy is nice (the look and feel equals the modeling tool), especially when filtering a large number of elements in huge models.
- That is a big advantage in big projects and product lines where several patterns are applied multiple times.
  An example might be a SOA project realizing 100 times the same transaction management - then such a tool pays off.

c) Consistency
Tool demo: Automatic consistency checks between design models and made design decisions; Model evolution, violations, and correcting the binding.
What is the interviewee's opinion? How useful is that, how could that be used in practice?

- Consistency checks and similar analyses are one of the main reasons for using modeling tools instead of drawing tools.
  Everything in this area is most welcome.
- Cross-subdomain validation is good! (knowledge models and architectural design models)
- Three ways of fixing a binding look good, also the GUI and the integrated markers (icons).
- Ownership and controlling might be a problem in big teams.
- However, the devil is in the details!
- Error messages must be simple and constructive and must show the source of the error.
  Bad examples: 'Design is inconsistent.' or too long and complex error messages.
- Potential problem in practice concerning scalability:
  Developers might get overwhelmed by too much data, e.g. too many proposals of how an error could be resolved.
- Could be potential future research topic:
  - Classification of binding errors
  - Proposal of how a binding error could get fixed properly
  - Danger: developers always ignore binding errors

d) Overall Feedback

- Opinion about tool integration vs. separated tools:
  Instead of aiming at a single tool that incorporates the functionality of multiple domains, specialized and well integrated tools might be better suited.
- Concerning number of decisions:
  Although there are projects with hundreds or thousands of decisions, not all of them have to be documented in such a tool and also not bound to the design.

Part 4: Opinions about Design Decisions in the Future

- Visualization of decision is current research topic and often requested.
- Knowledge is power!
  Everyone can program, but being smarter and having a way to manage and quickly acquire knowledge is important to be competitive.
- CMMI (Capability Maturity Model Integration), for instance, requires software compliance; the consistency checks in this tool could support that.

D.2.2 Interview 2

Part 1: Assessment of the Interviewee

Education	M.Sc. equivalent
Roles	(Lead) Architect, Developer
UML experience	10 years
UML skills	7/10
Decision making experience	5 years

Part 2: The old Way

a) *What is your typical way of making design decisions? (general procedure, process)*

- Workshops with peer architects for important decisions which have a lot of impact.
- Peer-Reviews and meetings with colleagues, especially with other architects;
  usually in meetings, important: whiteboard.
  Rule: 3 people x 3 hours is more efficient than e.g. +14h alone, because there is already a small review.
- Design decisions are either not documented or implicitly stored in the design document;
  Rarely explicit documentation of decisions.
  However, many architects in our company use a specific template for the documentation of decisions.
- Alternative solutions and rationale is often missing in documentation.

b) *What is your typical way of looking up made decisions / documentation?*

- If the responsible person is available, asking personally is the preferred way of retrieving the information.
- Otherwise, if detailed fine-grained decisions on code-level need to be looked up: Javadoc and possibly supporting code comments. It is often the best thing to keep low-level documentation "near" to the source code, because it is much easier to maintain and keep up-to-date.
- For higher-level artifacts, e.g. models in model-driven development, documentation is annotated as comments.
- Problem: if the documentation is only on the higher level, it might not get updated when the corresponding code changes.

c) *What is your typical way of realizing decisions in the design (model)?*

- Ideally, the architect works closely together with the developers.
- After delegating the work to the developers, reviews are used to check the realizations;
  However, that is critical, e.g. because the reviews might be incomplete due to time pressure.
- Having the architectural decisions in the model would be an improvement.
- An example for a decision is coding guidelines;
  can be assured with style checkers when committing code to the repository.
- Solutions of design decisions are often communicated via example implementations to developers;
  Examples are the best way to communicate particular realizations.

d) *What is your typical way of keeping your design (model) consistent with decisions / documentation / requirements?*

- Typically peer-reviews by other team members but also by external people!
- If possible, add documentation and define constraints in the models
- These reviews should be at least after each phase in the project, e.g. macro design, micro design, etc.
- In small teams up to 10 people: agile development with peer-work and peer-reviews

Part 3: Intention of this Research Project

a) Integrated documentation
Tool demo: Design decisions in the modeling tool – Issues, Alternatives, Outcomes, and Realizations.
What is the interviewee's opinion? How important is that information in practice?

- Realizations would be 'known uses'.
- It is important to model store your own realizations.
- It's very good if the architects can refer to and use predefined and standardized realizations.
- The (graphical) visualization of dependencies between the issues and alternatives is very important;
  reason: communication of decisions is much easier and understandable if decisions are represented graphically.

b) Automation of recurring work
Tool demo: Recurring changes in the design model are captured as best practices and can be reused.
What is the interviewee's opinion? How useful is that in practice?

- "I have drawn exactly the same once, but just as a design document. That requires that the developers completely understand it. The way you have shown it, we can enforce even more." (41:50)
- The biggest problem I see here is the education and discipline of the developers.
  This task has high requirements on the users.
- "Impressive concepts" (44:25), "a great evolution in Software Development" (45:20).

c) Consistency
Tool demo: Automatic consistency checks between design models and made design decisions; Model evolution, violations, and correcting the binding.
What is the interviewee's opinion? How useful is that, how could that be used in practice?

- It is hard to keep common structures, e.g. particular patterns, in different sub-projects consistent with each other.

D.2.3 Interview 3

Part 1: Assessment of the Interviewee

Education	M.Sc. equivalent
Roles	Developer
UML experience	4 years
UML skills	6/10
Decision making experience	None.

Part 2: The old Way

a) *What is your typical way of making design decisions? (general procedure, process)*

- Documents: requirements documentation, design/architecture documentation
- One main design document for internal development and also discussions with customers
- Decisions are typically documented in attached sections
- Solution architect is responsible for the project and severe decisions.
- Change request are used to decide issues together with the customer.

b) *What is your typical way of looking up made decisions / documentation?*

- As a developer: asking the solution architect who might delegate the questions to other responsible developers
- The documentation is rarely used because it rather describes the resulting design.
- Conservative software development does typically not make use of models.

c) *What is your typical way of realizing decisions in the design (model)?*

- The solution architect typically explains design decision to the developers who realize them in the design.
- Depending on the decision, either the solution architect or the developer documents the design in the main document.

- There are base product solutions of which a project uses a specialization for concrete customers.
  Well-proven designs or design fragments will be integrated back into the product solution.
  So the product solution contains the best practices.

d) *What is your typical way of keeping your design (model) consistent with decisions / documentation / requirements?*

- Quality assurance phases
- The individual development and coding tasks are relatively simple, inconsistencies are unlikely.

Part 3: Intention of this Research Project

a) Integrated documentation
Tool demo: Design decisions in the modeling tool – Issues, Alternatives, Outcomes, and Realizations.
What is the interviewee's opinion? How important is that information in practice?

- The knowledge database and especially the realizations are similar to what is known as domain development in model-driven development, in contrast to application development which targets the actual application;
  the required meta models of the domain (including DSLs) are specified – that might include templates that are similar to such realizations.

b) Automation of recurring work
Tool demo: Recurring changes in the design model are captured as best practices and can be reused.
What is the interviewee's opinion? How useful is that in practice?

- Concerning teaching that task is plausible.

c) Consistency
Tool demo: Automatic consistency checks between design models and made design decisions; Model evolution, violations, and correcting the binding.
What is the interviewee's opinion? How useful is that, how could that be used in practice?

- Error messages must be simpler to make it usable.

d) Overall Feedback

- Education of developers is crucial:
  - Many developers are not familiar with model-based development
  - Many developers do not have the time to learn about new concepts
  - The acquaintance of know how is not reasonable today
- Concerning the thesis:
  - The concepts are extremely reasonable, this is exactly where we want to go
  - However, not plausible in this company at the moment
  - The understanding of model differences is not trivial, teaching of developers required and the biggest problem!
- Skeptical concerning scalability (number of decisions and templates)
- In case this tool is used, a stepwise migration is required;
  some decisions are made and documented traditionally, some are made in the concepts including all benefits the tool offers.
- Also a problem: political discussions, new frameworks cannot easily be established.
  As soon as a tool has any weak point, it is out of consideration.
  Because of that, the tool cannot be used in practice (in our company).

D.2.4 Interview 4

Part 1: Assessment of the Interviewee

Education	Ph.D. and Junior Prof
Roles	Architect for academic tool
UML experience	15 years
UML skills	10/10
Decision making experience	10 years

Part 2: The old Way

a) *What is your typical way of making design decisions? (general procedure, process)*

- In a preparation phase, decisions and possible solutions are collected and evaluated.
- The decisions are, depending on the severity, made in meetings or in dedicated groups.
- Decisions are documented in meeting minutes and a Wiki, including pros/-cons.
- The results of metamodel related decisions are also documented directly in the metamodel.

b) *What is your typical way of looking up made decisions / documentation?*

- Mostly asking other developers.
- Information should be in the wiki, and other documents, but asking is faster and easier.

c) *What is your typical way of realizing decisions in the design (model)?*

- Developers (students, and PhD students) realizing decisions typically participate in the meetings.
- Assignment and documentation of progress of realizations tasks in Bugzilla.
- The communication between architect and developers is direct.

d) *What is your typical way of keeping your design (model) consistent with decisions / documentation / requirements?*

- Since parts of the code is generated from the metamodel, the generation ensures consistency between these two artifacts.
- Several design decisions are documented but not realized!
- Consistency is checked manually by groups of 5–6 developers in a design review, sometimes including the architect.

Part 3: Intention of this Research Project

a) Integrated documentation
Tool demo: Design decisions in the modeling tool – Issues, Alternatives, Outcomes, and Realizations.

What is the interviewee's opinion? How important is that information in practice?

- That is familiar, we prepared decisions in a similar way, based on Kruchten's ontology.
- We described the variability in realizations with feature diagrams.
- I hope that reuse of decisions will be important in the future.
- The tooling is important!
- Integration of decision management with other tools like DOORS, Bugzilla, etc. is very important!

b) Automation of recurring work
Tool demo: Recurring changes in the design model are captured as best practices and can be reused.
What is the interviewee's opinion? How useful is that in practice?

- Would be nice to see the delta directly in the model before applying realizations.
- Reasonable and sensible in development of product-lines or the like.
- Looks very similar to UML collaborations!
  The result of decisions might be represented in UML collaborations in the model as an easy way of documentation within the model.
- Representation is understandable for scientists but not usable in practice. If used in practice, the matching step must be visible on the model-level, not on the metamodel-level.
- In the context of specific domains like product lines, this feature is highly reasonable.
- Might not be reasonable in behavioral models because realizations in behavioral models might be very specific and not applicable in many other contexts.
- Recording of changes is basic requirement – otherwise, no one will do it!

c) Consistency
Tool demo: Automatic consistency checks between design models and made design decisions; Model evolution, violations, and correcting the binding.
What is the interviewee's opinion? How useful is that, how could that be used in practice?

- Reasonable for small set of changes, test of practicability for bigger sets

of changes required.

d) Overall Feedback

- Useful in structural diagrams like class- or component diagrams.
- Not that reasonable for other types of models, like behavioral diagrams. I don't see many reusable parts e.g. in sequence or activity diagrams.
- UML collaboration diagrams could be used to visualize design decisions in UML models.

D.2.5 Interview 5

Part 1: Assessment of the Interviewee

Education	M.Sc. equivalent
Roles	Modeler (also other roles because of small projects)
UML experience	5 years, academic skills only
UML skills	8/10
Decision making experience	None – only implicit decision making in small projects.

Part 2: The old Way

a) *What is your typical way of making design decisions? (general procedure, process)*

- Decisions have only been made in small web application projects, in particular the choice of a base framework.
- The choice of the solution was made after evaluating project-specific requirements (pros/cons).
- The choice was an implicit choice, no dedicated documentation exists.
- The only documentation are emails and maybe meeting minutes.
- Documentation concerning design is only made partially in form of Javadoc.

b) *What is your typical way of looking up made decisions / documentation?*

- Asking the responsible developer in person, e.g. by phone.

- Reading Javadoc.

c) *What is your typical way of realizing decisions in the design (model)?*

- The decision maker realizes the decisions.

d) *What is your typical way of keeping your design (model) consistent with decisions / documentation / requirements?*

- Documentation did not exist in these small projects, neither requirement nor design specification.
- Many consistency checks were performed manually.
- Automated unit tests.

Part 3: Intention of this Research Project

a) Integrated documentation
Tool demo: Design decisions in the modeling tool – Issues, Alternatives, Outcomes, and Realizations.
What is the interviewee's opinion? How important is that information in practice?

- The integration of both tools is great.
- Could be used for guiding the developers through design decisions, might be similar to Mylyn.
- Dependencies between design decisions are important.
- Keeping project-specific and project-independent parts separate is important to enable reuse.

b) Automation of recurring work
Tool demo: Recurring changes in the design model are captured as best practices and can be reused.
What is the interviewee's opinion? How useful is that in practice?

- That application of realizations to design models is a reasonable/sensible thing.

- That is a systematic procedure.
- The presentation might be technical, but it is not a problem because pretty printer could take care of that.
- On-the-fly decision capturing is well thought-out.
- (If parameterization of model-independent differences would be possible:) Ontologies may be useful in companies to automatically apply company-specific customizations.

c) Consistency
Tool demo: Automatic consistency checks between design models and made design decisions; Model evolution, violations, and correcting the binding.
What is the interviewee's opinion? How useful is that, how could that be used in practice?

- The validation is super.
- But only starting at a certain size of models.
- This could be a nice tool for easing realizations and validation of decisions and for making both tasks faster.
- Custom constraints could be very useful for variants of realizations.
- The validation would very well complement unit tests.

D.2.6 Interview 6

Part 1: Assessment of the Interviewee

Education	M.Sc. equivalent
Roles	Architect and Modeler in academic context
UML experience	8 years
UML skills	10/10
Decision making experience	None – only few implicit decisions so far.

Part 2: The old Way

a) *What is your typical way of making design decisions? (general procedure, process)*

- Severe decisions are made in meetings, usually with a whiteboard.

- Design issues are discussed ad-hoc, e.g. with examples and having parts of the system running.
- Decisions and design are documented in meeting minutes, Javadoc, and annotations directly in the model.

b) *What is your typical way of looking up made decisions / documentation?*

- That is an important problem!
- Mostly reading Javadoc and model annotations.
- Preferably reading and debugging the system; asking the developers (students) was not very successful.
- There was no separate design documentation in the work by students, only a document describing the high-level architecture.

c) *What is your typical way of realizing decisions in the design (model)?*

- The developer realizing a decision participates in the meetings in which the decisions are made.

d) *What is your typical way of keeping your design (model) consistent with decisions / documentation / requirements?*

- Test cases!
- Comparing the system's behavior with the expected behavior specified in use cases (manual testing).

Part 3: Intention of this Research Project

a) Integrated documentation
Tool demo: Design decisions in the modeling tool – Issues, Alternatives, Outcomes, and Realizations.
What is the interviewee's opinion? How important is that information in practice?

- Integration is nicely done!
- However, the overhead is not worth the effort in small projects with 1–2 developers.

- Mylyn also integrates into Eclipse, maybe it can be integrated, too.

b) Automation of recurring work
Tool demo: Recurring changes in the design model are captured as best practices and can be reused.
What is the interviewee's opinion? How useful is that in practice?

- In principle a cool feature.
- But it requires that the user knows how the feature works! In particular what the context is and how it can be defined.
- Concrete (graphical) syntax for model changes would help a lot to understand realizations.

c) Consistency
Tool demo: Automatic consistency checks between design models and made design decisions; Model evolution, violations, and correcting the binding.
What is the interviewee's opinion? How useful is that, how could that be used in practice?

- It must be possible to overwrite or reject decisions.
- It is quite cumbersome to manually adjust the binding.

d) Overall Feedback

- Capturing decisions on-the-fly:
  The capturing feature is very useful for architects who also use the role for realizing the decisions.
- Probably problems when multiple developers are working on the same model and binding.
- An interesting evaluation would be the following scenario:
  - An architect models 10 decisions incl. their realizations
  - Another developer should then apply these decisions to a model.

Bibliography

[AP03] Marcus Alanen and Ivan Porres. Difference and Union of Models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 – The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, October 2003.

[Atl08] ATL (M2M/Atlas Transformation Language). http://www.eclipse.org/atl/, 2010.

[BB04] Janet E. Burge and David C. Brown. An integrated Approach for Software Design Checking Using Design Rationale. In *1st International Conference on Design Computing and Cognition (DCC '04)*, pages 557–576. Kluwer Academic Press, 2004.

[BB08] Janet Burge and David Brown. Seurat: Integrated Rationale Management. In *Proceedings of the 30th International Conference on Software Engineering*, pages 835–838, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[BDLvV09] Muhammad Ali Babar, Torgeir Dingsøyr, Patricia Lago, and Hans van Vliet. *Software Architecture Knowledge Management: Theory and Practice*. Springer, 1st edition, August 2009. ISBN: 978-3-642-02373-6.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[BGJ05] Muhammad Ali Babar, Ian Gorton, and D. Ross Jeffery. Capturing and using software architecture knowledge for architecture-

based software development. In *Evolvable Hardware*, pages 169–176. IEEE Computer Society, 2005.

[BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, January 2005.

[BLS+09] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. An Example is Worth a Thousand Words: Composite Operation Modeling By-Example. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 2009.

[BM05] Felix Bachmann and Paulo Merson. Experience Using the Web-Based Tool Wiki for Architecture Documentation. Technical Report CMU/SEI-2005-TN-041, Carnegie Mellon University, Software Engineering Institute, September 2005.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley, 1998.

[BP08] Cédric Brun and Alfonso Pierantonio. Model Differences in the Eclipse Modelling Framework. *CEPIS Upgrade – The European Journal for the Informatics Professional*, IX(2):29–34, April 2008.

[BWG05] Muhammad Ali Babar, Xiaowen Wang, and Ian Gorton. PAKME: A Tool for Capturing and Using Architecture Design Knowledge. In *9th International Multitopic Conference, IEEE INMIC 2005*, pages 1–6, December 2005.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CND07] Rafael Capilla, Francisco Nava, and Juan C. Duenas. Modeling and Documenting the Evolution of Architectural Design Decisions. In *SHARK-ADI '07: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, pages 9–15, Washington, DC, USA, 2007. IEEE Computer Society.

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[dBFL+07] Remco C. de Boer, Rik Farenhorst, Patricia Lago, Hans van Vliet, Viktor Clerc, and Anton Jansen. Architectural knowledge: Getting to the Core. In *Proceedings of the Quality of Software Architectures 3rd International Conference on Software Architectures, Components, and Applications*, QoSA '07, pages 197–214, Berlin, Heidelberg, 2007. Springer-Verlag.

[Dij07] Remco Dijkman. A Classification of Differences between Similar BusinessProcesses. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, pages 37–50, Washington, DC, USA, 2007. IEEE Computer Society.

[DKPF08] Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. Engineering a DSL for Software Traceability. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *SLE*, volume 5452 of *Lecture Notes in Computer Science*, pages 151–167. Springer, 2008.

[Ecl04] Eclipse Foundation. Eclipse Public License. http://www.eclipse.org/legal/epl-v10.html, February 2004. v1.0.

[emf10] EMF Compare project. http://www.eclipse.org/modeling/emf/?project=compare, 2011.

[Eys09] Moritz Eysholdt. EPatch. http://wiki.eclipse.org/EMF_Compare/Epatch, 2010.

[FBFG07] Franck Fleurey, Benoit Baudry, Robert B. France, and Sudipto Ghosh. A Generic Approach for Automatic Model Composition. In Holger Giese, editor, *MoDELS*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer, 2007.

[FCKK01] Davide Falessi, Giovanni Cantone, Rick Kazman, and Philippe Kruchten. Decision-making techniques for software architecture design: a comparative survey. *ACM Computing Surveys*, in press.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 157–167. Springer Berlin / Heidelberg, 2000.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, June 1999. With contributions by Kent Beck, John Brant, Willima Opdyke, and Don Roberts.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, Reading, Massachusetts, November 2002.

[FW07] Sabrina Förtsch and Bernhard Westfechtel. Differencing and Merging of Software Diagrams – State of the Art and Challenges. In Joaquin Filipe, Markus Helfert, and Boris Shishkov, editors, *International Conference on Software and Data Technologies (ICSOFT), Setubal (Portugal)*, volume 2, pages 90–99. Institute for Systems and Technologies for Information, Control and Communication, July 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1st edition, January 1995.

[GKLE10] Christian Gerth, Jochen Malte Küster, Markus Luckey, and Gregor Engels. Precise Detection of Conflicting Change Operations Using Process Model Terms. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems – 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, volume 6395 of *Lecture Notes in Computer Science*, pages 93–107. Springer, October 2010.

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[GWO09] James R. Groff, Paul N. Weinberg, and Andrew J. Oppel. *SQL: The Complete Reference*. McGraw-Hill Osborne Media, third edition, August 2009.

[HAZ07] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. Using Patterns to Capture Architectural Decisions. *IEEE Software*, 24(4):38–45, 2007.

[HET08] Frank Hermann, Hartmut Ehrig, and Gabriele Taentzer. A Typed Attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams. Electron. Notes Theor. Comput. Sci., Elsevier Science Publishers B. V., 211, pages 261–269, 2008.

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual

Syntax for Models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA '09)*, pages 114–129. Springer-Verlag, 2009.

[HK10] Markus Herrmannsdoerfer and Maximilian Koegel. Towards a Generic Operation Recorder for Model Evolution. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 76–81. ACM, New York, NY, USA, 2010.

[IBM10] IBM. IBM Rational DOORS. http://www.ibm.com/software/awdtools/doors/, 2010.

[JB05] Anton Jansen and Jan Bosch. Software Architecture as a Set of Architectural Design Decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA '05)*, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society.

[JvdVAH07] Anton Jansen, Jan van der Ven, Paris Avgeriou, and Dieter K. Hammer. Tool Support for Architectural Decisions. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '07)*, pages 44-53. IEEE Computer Society, 2007.

[KBAW94] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. In Bruno Fadini, editor, *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*, pages 81–90, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[KGFE08] Jochen Küster, Christian Gerth, Alexander Förster, and Gregor Engels. Detecting and Resolving Process Model Differences in the Absence of a Change Log. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, pages 244–260. Springer Berlin / Heidelberg, 2008.

[KK03] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[KKC00] Rick Kazman, Mark H. Klein, and Paul C. Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, August 2000.

[KLvV06] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building Up and Reasoning About Architectural Knowledge. In hristine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Sec-*

ond International Conference on Quality of Software Architectures, QoSA 2006, volume 4214 of *LNCS*, pages 43–58. Springer, 2006.

[Kol09] Dimitrios S. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2009.

[Kön09a] Patrick Könemann. Integrating a Design Decision Management System with a UML Modeling Tool. Technical Report IMM-Technical Report-2009-07, Technical University of Denmark, April 2009.

[Kön09b] Patrick Könemann. Integrating Decision Management with UML Modeling Concepts and Tools. In *WICSA/ECSA*, pages 297–300. IEEE Computer Society, September 2009.

[Kön09c] Patrick Könemann. Model-Independent Differences. *ICSE Workshop on Comparison and Versioning of Software Models*, 0:37–42, May 2009.

[Kön10a] Patrick Könemann. Capturing the Intention of Model Changes. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems – 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, volume 6395 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2010.

[Kön10b] Patrick Könemann. Semantic Grouping of Model Changes. In *Proceedings of the 1st International Workshop on Model Comparison in Practice (IWMCP '10)*, pages 50–55, ACM, New York, NY, USA, July 2010.

[KPP06a] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging Models with the Epsilon Merging Language. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer, October 2006.

[KPP06b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Object Language (EOL). In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.

[KPRP07] Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A. C. Polack. Bridging the Epsilon Wizard Language and the Eclipse Graphical Modeling Framework. In Modeling Symposium, Eclipse Summit Europe, October 2007.

[KRPP09] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. *ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, IEEE Computer Society, May 2009.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Kru04] Philippe Kruchten. An Ontology of Architectural Design Decisions in Software Intensive Systems. In *2nd Groningen Workshop Software Variability*, pages 54–61, October 2004.

[KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. A Generic Difference Algorithm for UML Models. In Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 105–116. GI, 2005.

[KZ10] Patrick Könemann and Olaf Zimmermann. Linking Design Decisions to Design Models in Model-based Software Development. In Muhammad Ali Babar and Ian Gorton, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23–26*, volume 6285 of *Lecture Notes in Computer Science*, pages 246–262, Copenhagen, Denmark, August 2010. Springer.

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems*, 16(4):349–361, August 2007. Special Issue on Model-Driven Systems Development.

[LJA09] Peng Liang, Anton Jansen, and Paris Avgeriou. Knowledge Architect: A Tool Suite for Managing Software Architecture Knowledge. Technical Report RUG-SEARCH-09-L01, University of Groningen, February 2009.

[LJA10] Peng Liang, Anton Jansen, and Paris Avgeriou. Collaborative Software Architecting through Knowledge Sharing. In Ivan Mistrík, John Grundy, André Hoek, and Jim Whitehead, editors, *Collaborative Software Engineering*, pages 343–367. Springer Berlin Heidelberg, 2010.

[LK08a] Larix Lee and Philippe Kruchten. Customizing the Capture of Software Architectural Design Decisions. In *IEEE Canadian Conference on Electrical and Computer Engineering, CCECE 2008*, pages 693–698. British Columbia Univ., Vancouver, BC, 2008.

[LK08b] Larix Lee and Philippe Kruchten. A Tool to Visualize Architectural Design Decisions. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *4th International Conference on the Quality of Software-Architectures, QoSA 2008, Karlsruhe, Germany, October 14-17*, volume 5281 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2008.

[LR99] Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, September 1999.

[MCH10] Patrick Mäder and Jane Cleland-Huang. A Visual Traceability Modeling Language. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *MoDELS (1)*, volume 6394 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2010.

[Men02] Tom Mens. A State-of-the-art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[OMG06a] Object Management Group. Meta Object Facility (MOF) Core Specification. http://www.omg.org/cgi-bin/doc?formal/2006-01-01, January 2006. Version 2.0.

[OMG06b] Object Management Group. Object Constraint Language Specification, Version 2.0. http://www.omg.org/cgi-bin/doc?formal/2006-05-01, May 2006.

[OMG07a] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/cgi-bin/doc?ptc/2007-07-07, July 2007.

[OMG07b] Object Management Group. MOF 2.0/XMI Mapping. http://www.omg.org/spec/XMI/2.1.1/, December 2007. Version 2.1.1.

[OMG09] Object Management Group. Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/1.2, March 2009.

[OMG10] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.3*. http://www.omg.org/cgi-bin/doc?formal/2010-05-05, May 2010.

[OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, New York, NY, USA, 2003. ACM Press.

[Par94] David Lorge Parnas. Software Aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[RHW+10] Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitrios S. Kolovos, Kelly Garcés, Richard F. Paige, and Fiona A. C. Polack. A Comparison of Model Migration Tools. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2010.

[RSA10] Jan Reimann, Mirko Seifert, and Uwe Aßmann. Role-based Generic Model Refactoring. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6395 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2010.

[SK08] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer Berlin / Heidelberg, 2008.

[SLK09] Mojtaba Shahin, Peng Liang, and Mohammad Reza Khayyambashi. Architectural Design Decision: Existing Models and Tools. In *WICSA*. IEEE Computer Society, September 2009.

[SZ05] G. Spanoudakis and A. Zisman. Software Traceability: A Roadmap. In S. K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, volume 3. World Scientific, August 2005.

[TA05] Jeff Tyree and Art Akerman. Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22(2):19–27, 2005.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference Computation of Large Models. In *ESEC-FSE '07: Foundations of Software Engineering*, pages 295–304, Dubrovnik, Croatia, September 2007. ACM.

[TDSF10] Chouki Tibermacine, Christophe Dony, Salah Sadou, and Luc Fabresse. Software Architecture Constraints as Customizable, Reusable and Composable Entities. In Muhammad Ali Babar and Ian Gorton, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23–26*, volume 6285 of *Lecture Notes in Computer Science*, pages 505–509, Copenhagen, Denmark, August 2010. Springer.

[Tea10] CMMI Product Team. CMMI for Development, version 1.3. Technical Report CMU/SEI-2010-TR-033, Software Engineering Institute, Carnegie Mellon University, November 2010.

[TAJ+10] Antony Tang, Paris Avgeriou, Anton Jansen, Rafael Capilla and Muhammad Ali Babar. A Comparative Study of Architecture Knowledge Management Tools. *Journal of Systems and Software*, 83(3):352–370, March 2010.

[TJH07] Antony Tang, Yan Jin, and Jun Han. A Rationale-based Architecture Model for Design Traceability and Reasoning. *Journal of Systems and Software*, 80(6):918–934, June 2007.

[uhB09] Aman ul haq and Muhammad Ali Babar. Tool Support for Automating Architectural Knowledge Extraction. In *Proceedings of the 2009 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, SHARK '09, pages 49–56, Washington, DC, USA, 2009. IEEE Computer Society.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[Var06] Dániel Varró. Model Transformation by Example. In *MoDELS*, pages 410–424, 2006.

[Ven99] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, second edition, 1999.

[Wag08] Robert Wagner. *Inkrementelle Modellsynchronisation*. Dissertationen, University of Paderborn, 2008.

[Wah08] Michael Wahler. *Using Patterns to develop Consistent Design Constraints*. PhD thesis, Technische Hochschule ETH Zürich, 2008.

[XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 54–65. ACM, 2005.

[ZDGH05] Olaf Zimmermann, Vadim Doubrovski, Jonas Grundler, and Kerard Hogg. Service-oriented Architecture and Business Process Choreography in an Order Management Scenario: Rationale, Concepts, Lessons Learned. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–312, New York, NY, USA, 2005. ACM.

[ZGK+07] Olaf Zimmermann, Thomas Gschwind, Jochen Malte Küster, Frank Leymann, and Nelly Schuster. Reusable Architectural Decision Models for Enterprise Application Development. In Sven Overhage, Clemens A. Szyperski, Ralf Reussner, and Judith A. Stafford, editors, *QoSA*, volume 4880 of *Lecture Notes in Computer Science*, pages 15–32. Springer, July 2007.

[Zim09] Olaf Zimmermann. *An Architectural Decision Modeling Framework for Service-Oriented Architecture Design*. Dissertation, University of Stuttgart, 2009.

[Zim11] Olaf Zimmermann. Architectural Decisions as Reusable Design Assets. *IEEE Software*, 28:64–69, January/February 2011.

[ZKL+09] Olaf Zimmermann, Jana Koehler, Frank Leymann, R. Polley, and Nelly Schuster. Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. *Journal of Systems and Software*, 82(8):1249–1267, 2009. SI: Architectural Decisions and Rationale.

[ZKD+09] Steffen Zschaler, Dimitrios S. Kolovos, Nikolaos Drivalos, Richard F. Paige, and Awais Rashid. Domain-Specific Metamodelling Languages for Software Language Engineering. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *SLE*, volume 5969 of *Lecture Notes in Computer Science*, pages 334–353. Springer, 2009.